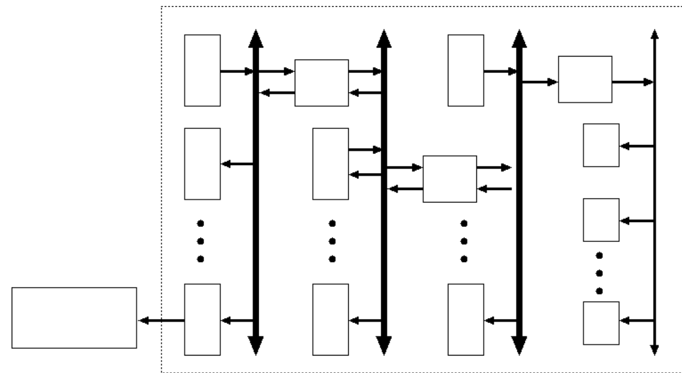
System on Chip
Design and Modelling



University of Cambridge
Computer Laboratory
Lecture Notes
(Final)

Dr. David J Greaves

(C) 2008-18 All Rights Reserved DJG.

Part II
Computer Science Tripos
Michaelmas Term 2017/18

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

0.0.1 SoC Design : 2017/18: Twelve Lectures for CST Part II

Over previous decades, most of the advances in computer performance have arisen from shrinking the physical size of the computer according to Moore's Law. But when we reached 100 nm transistor size, Dennard Scaling ceased and new computer architectures were required. Semiconductor physicists have provided a world where we can put much more logic on our System On Chip (SoC) that we can conveniently power up at once (Dark Silicon), meaning that application-specific accelerators are increasingly being used. How else does your mobile phone compress motion video without almost immediately flattening the battery?

In this course we examine the basic energy and performance metrics for todays chip multi-processors (CMPs), caches, busses, DRAM banks and custom accelerators and examine the need for, design of and integration of custom accelerators. We briefly visit all of the IP blocks found on a typical SoC, as used in the Raspberry Pi. We look at the future of reconfigurable computing and the role of FPGA in the datacentre.

Examples will assume knowledge of three languages, C, Verilog and assembly language but not require any degree of proficiency in these languages.

A system on a chip (SoC) consists of several different microprocessor subsystems together with memories and I/O interfaces and hardware accelerators. We shall cover the relevant design and modelling techniques leading to an overall emphasis on architectural exploration, where we can estimate energy and performance for a variety of designs without investing effort of making a detailed implementation. This is the "front end" of the design automation tool chain. (Back end material, such as design of individual gates, layout, routing and fabrication of silicon chips is not covered.)

[Note to supervisors: This year, Formal Methods and Assertion-based design are again not being lectured. Instead we have a new sections on Accelerator Structures and High-Level Synthesis. And, as per the last two years, the SystemC net-level facilities will not be lectured - we'll just do enough SystemC to cover TLM modelling in the ESL section.]

0.0.2 Recommended Reading

Subscribe for webcasts from 'Design And Reuse': www.design-reuse.com Ed Lipianski's Book

Embedded Systems Hardware for Software Engineers Ed Lipiansky. McGraw-Hill Professional (1 Feb. 2012)

Embedded Systems Hardware for Software Engineers describes the electrical and electronic circuits that are used in embedded systems, their functions, and how they can be interfaced to other devices.

Basic computer architecture topics, memory, address decoding techniques, ROM, RAM, DRAM, DDR, cache memory, and memory hierarchy are discussed. The book covers key architectural features of

widely used microcontrollers and microprocessors, including Microchip's PIC32, ATMEL's AVR32, and Freescale's MC68000. Interfacing to an embedded system is then described. Data acquisition system level design considerations and a design example are presented with real-world parameters and characteristics. Serial interfaces such as RS-232, RS-485, PC, and USB are addressed and printed circuit boards and high-speed signal propagation over transmission lines are covered with a minimum of math. A brief survey of logic families of integrated circuits and programmable logic devices is also contained in this in-depth resource.

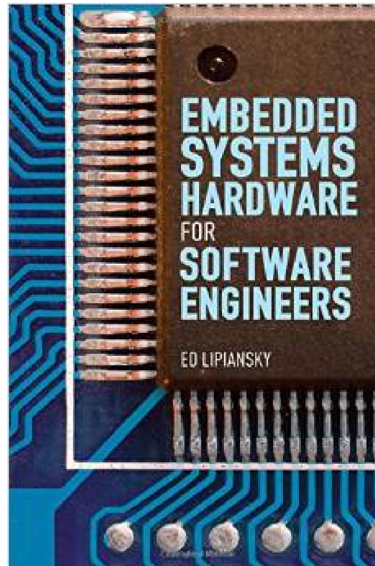


Figure 1: Embedded Systems Hardware for Software Engineers.

Ed Lipianski's Book

Mishra K. (2014) *Advanced Chip Design - Practial Examples in Verilog.* Pub Amazon Martson Gate.

Multicore field-programmable SoC: Xilinx Zync Product Brief

Atmel, ARM-based Embedded MPU AT91SAM Datasheet

OSCI. *SystemC tutorials and whitepapers* . Download from OSCI www.accelera.org or copy from course web site.

Brian Bailey, Grant Martin. *ESL Models and Their Application: Electronic System Level Design.* Springer.

The Simple Art of SoC Design - Closing the gap between ESL and RTL Michael Keating (2011), Springer ISBN 978-1-4419-8586-6

Ghenassia, F. (2006). *Transaction-level modeling with SystemC: TLM concepts and applications for embedded systems* . Springer.

Eisner, C. & Fisman, D. (2006). *A practical introduction to PSL* . Springer (Series on Integrated Circuits and Systems) — not relevant this year.

Foster, H.D. & Krolnik, A.C. (2008). *Creating assertion-based IP* . Springer (Series on Integrated Circuits and Systems) — not relevant this year.

Grotker, T., Liao, S., Martin, G. & Swan, S. (2002). *System design with SystemC* . Springer. E-BOOK (PDF)

Wolf, W. (2002). *Modern VLSI design (System-on-chip design)* . Pearson Education. Online: LINK.

0.0.3 Example: A Cellphone.

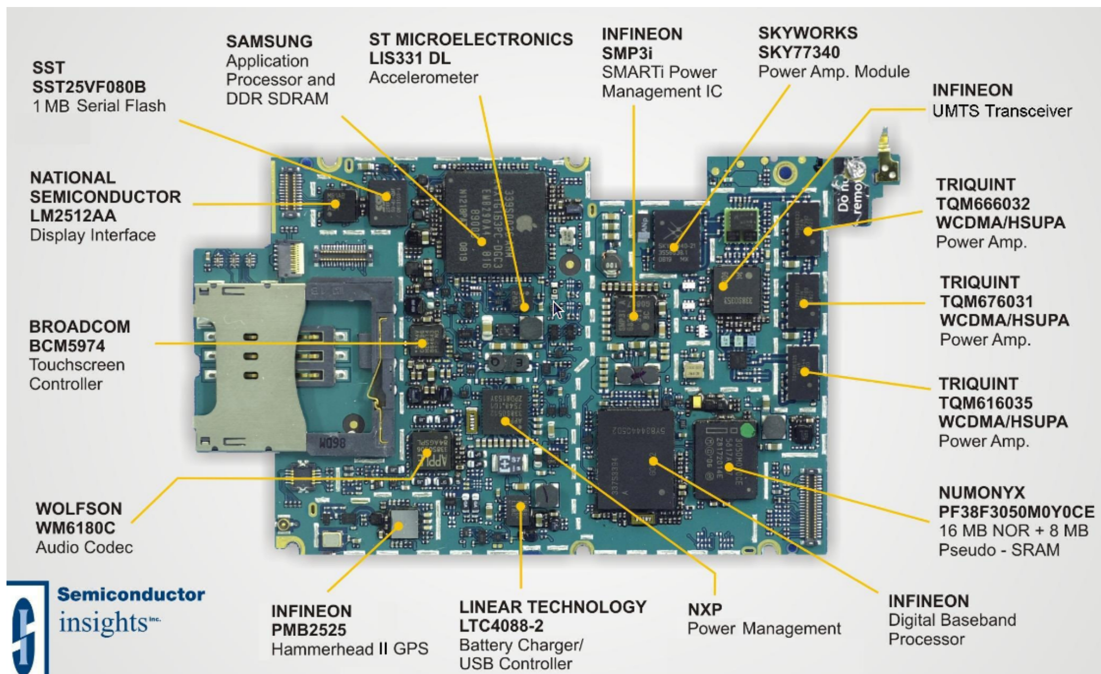


Figure 2: One of the two main PCBs of an Apple iPhone. Main SoC is top, left-centre.

A modern mobile phone contains eight or more radio tranceivers, counting the various cellphone standards, GPS, WiFi, near-field and Bluetooth. For the Apple iPhones, all use off-SoC mixers and some use on-SoC ADC/DAC. Another iPhone teardown link

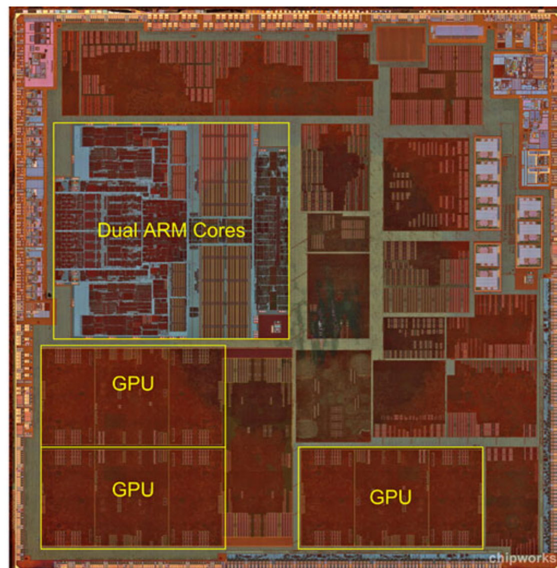


Figure 3: An Apple SoC - Two ARM and 3 GPU cores. Made by arch rival Samsung.

Further examples: iFixit Teardowns iPhone Dissected

Samsung GalaxyNumonyx Flash DatasheetCellphone physical components - bill of materials:

- Main SoC - Application Processor, Caches and **custom accelerators** for MP3 and MPEG and so on

- DRAM (dynamic random-access memory)
- Display (touch sensitive) + Keypad + Misc buttons
- Audio ringers and speakers, microphone(s) (noise cancelling),
- Infra-red IRDA port
- Multi-media codecs (A/V capture and replay in several formats)
- Radio Interfaces: GSM (three or four bands), BlueTooth, 802.11, GPS, Nearfield, ... plus antennas.
- Power Management: Battery Control, Processor Speed, on/off/flight modes.
- Front and Rear Cameras, Flash/Torch and ambient light sensor,
- Memory card slot,
- Physical connectors: USB, Power, Headset,
- Case, Battery and PCBs
- Java VM and Operating System.

0.0.4 Introduction: What is a SoC (1/2)?

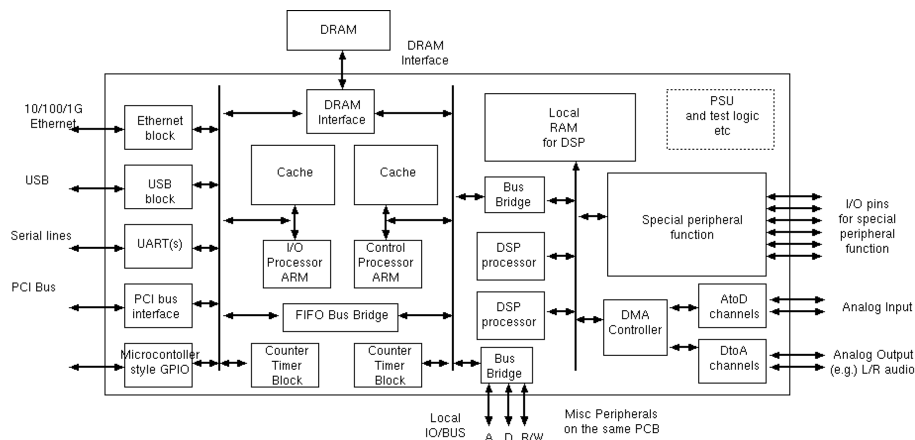


Figure 4: Block diagram of a multi-core 'platform' chip, used in a number of networking products. Programmable DSP processesors were used instead of hard-wired accelerators.

A System On A Chip: typically uses 70 to 140 mm^2 of silicon.

Multicore field-programmable SoC Xilinx Product Brief: PDFAtmel ARM-Based Platform Chip: PDF

0.0.5 Introduction: What is a SoC (2/2)?

A SoC is a complete system on a chip. A 'system' includes a microprocessor, memory and peripherals. The processor may be a custom or standard microprocessor, or it could be a specialised media processor for sound, modem or video applications. There may be multiple processors and also other generators of bus cycles, such as DMA controllers. DMA controllers can be arbitrarily complex and are only really distinguished from processors by their complete or partial lack of instruction fetching.

Processors are interconnected using a variety of mechanisms, including shared memories and message-passing hardware entities such as general on-chip networks and specialised channels and mailboxes.

SoCs are found in every consumer product, from modems, mobile phones, DVD players, televisions and iPods.

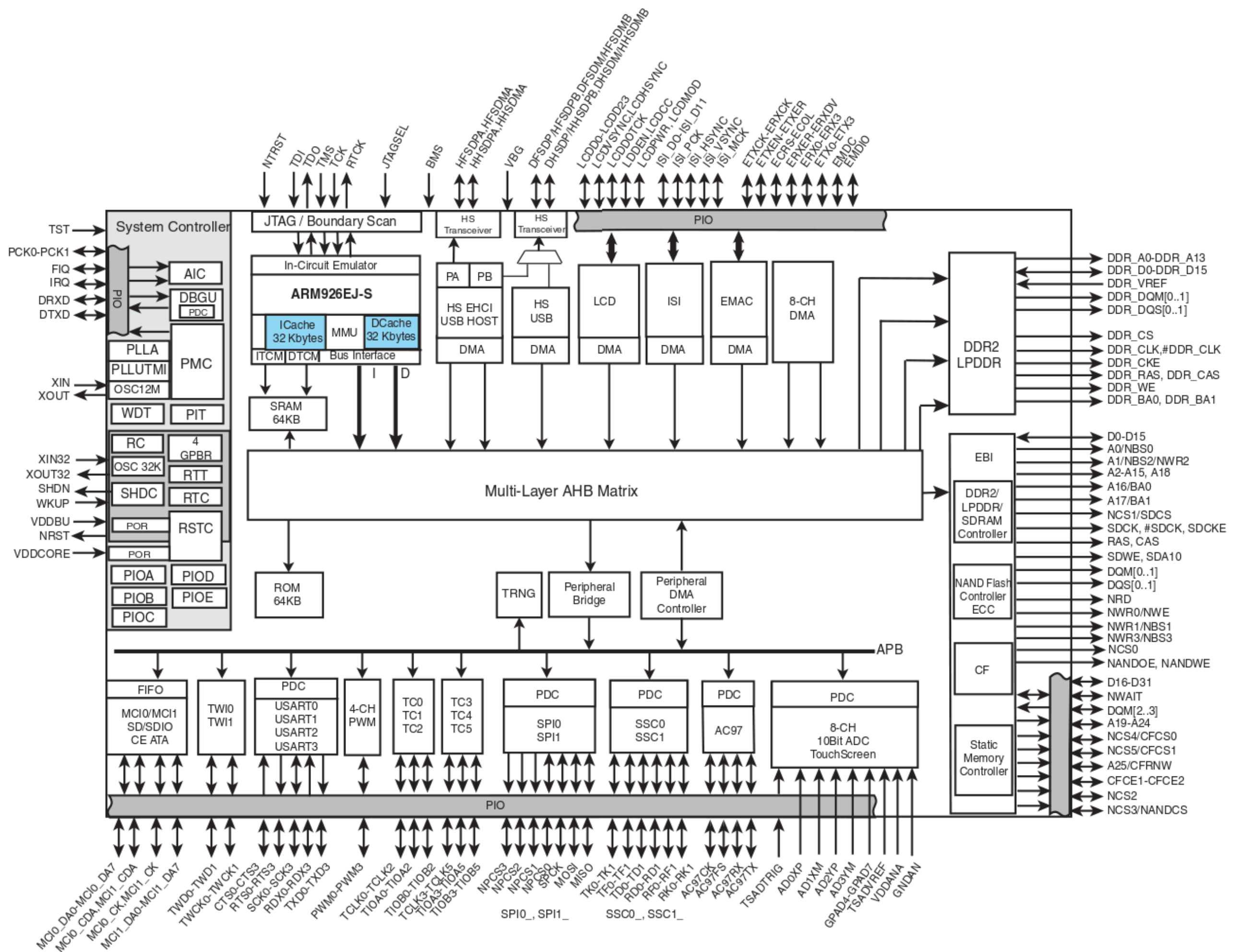


Figure 5: Platform Chip Example: Atmel SAM Series 9645.

0.1 Hardware Design Flow for an ASIC

ASIC: Application-Specific Integrated Circuit. The ASIC hardware design flow is divided by the **Structural RTL** level into:

- **Front End**: specify, explore, design, capture, synthesise ⇝ **Structural RTL**
- **Back End**: **Structural RTL** ⇝ place, route, mask making, fabrication.

There is a companion software design flow that must mesh perfectly with the hardware if the final product is to work first time. This course will put as much emphasis on field-programmable parts (FPGAs) as on ASICs, since FPGA has now grown in importance.

Figure 6 shows a typical design and maufacturing flow that leads from design capture to ASIC fabrication. This might take six months. The FPGA equivalent can be done in half a day!

0.1.1 Front End

The design must be specified in terms of high-level requirements, such as function, throughput and power consumption.

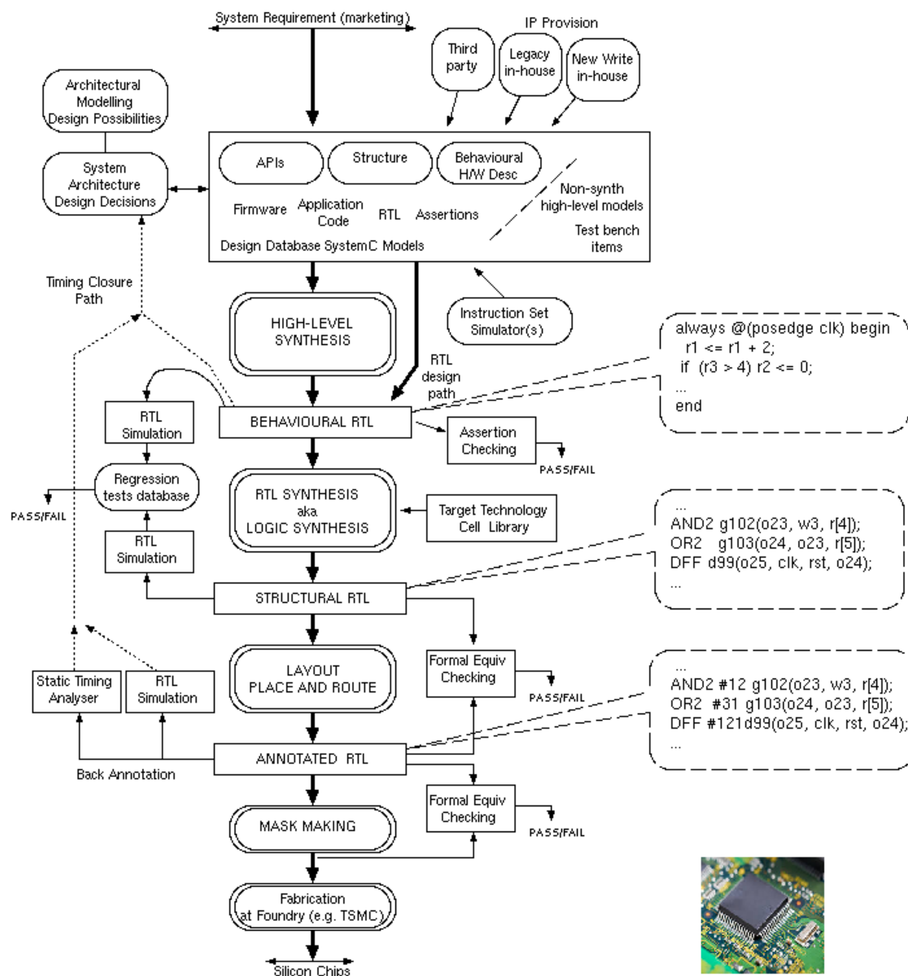


Figure 6: Design and Manufacturing Flow for SoC.

Design capture: it is transferred from the marketing person's mind, back of envelope or or wordprocessor document into machine-readable form.

Architectural exploration will try different combinations of processors, memories and bus structures to find an implementation with good power and load balancing. A loosely-timed high-level model is sufficient to compute the performance of an architecture.

Detailed design will select IP (interlectual property) providers for all of the functional blocks, or else they will exist from previous in-house designs and can be used without license fees, or else freshly written.

Logic synthesis will convert from behavioural RTL to structural RTL. Synthesis from formal high-level forms, including C,C++, SysML statecharts, formal specifications of interfaces and behaviour is beginning to be used.

Instruction set simulators (ISS) for embedded processors are needed: purchased from third parties such as ARM and MIPS, or as a by-product of custom processor design.

The interface specifications (register maps and other APIs) between components need to be stored: the IP-XACT format may be used.

High-level models that are never intended to be synthesisable and test bench components will also be coded, typically using SystemC.

0.1.2 Back End

After RTL synthesis using a target technology library, we have a structural netlist that has no gate delays. Place and route gives 2-D co-ordinates to each component, adds external I/O pads and puts wiring between the components. RTL annotated with actual implementation gate delays gives a precise power and performance model. If performance is not up to par, design changes are needed.

Fabrication of masks is commonly the most expensive single step (e.g. one million pounds), so must be correct first time.

Fabrication is performed in-house by certain large companies (e.g. Intel, Samsung) but most companies use foundaries (UMC, TSMC).

At all stages (front and back end), a library of standard tests will be run every night and any changes that cause a previously-passing test to fail (regressions) will be automatically reported to the project manager.

0.1.3 Levels of Modelling Abstraction

Our modelling system must support all stages of the design process, from design entry to fabrication. But we cannot model a complete SoC in detail and expect to simulate the booting of the O/S in reasonable time. We typically use a model called an **ESL Virtual Platform**. And where we are interested in the details of a specific module, we need to mix components using different levels of modelling abstraction within a single virtual platform.

Levels commonly used are:

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

Other terms in use are:

- **Programmer View Accurate:** The contents of visible memory and registers is as per the real hardware, but timing may be inaccurate and other registers or combinational nets that are not designated as part of the 'programmers view' may not be modelled accurately.
- **Behavioural Modelling:** Using a threads package, or other library (e.g. SystemC), hand-crafted programs are written to model the behaviour of each component or subsystem. Major hardware items such as busses, caches or DRAM controllers may be neglected in such a model.

The Programmer's View is often abbreviated as 'PV' and if timing is added it is called 'PV+T'.

The Programmer's View contains only architecturally-significant registers such as those that the software programmer can manipulate with instructions. Other registers in a particular hardware implementation, such as pipeline stages and holding registers to overcome structural hazards, are not part of the PV.

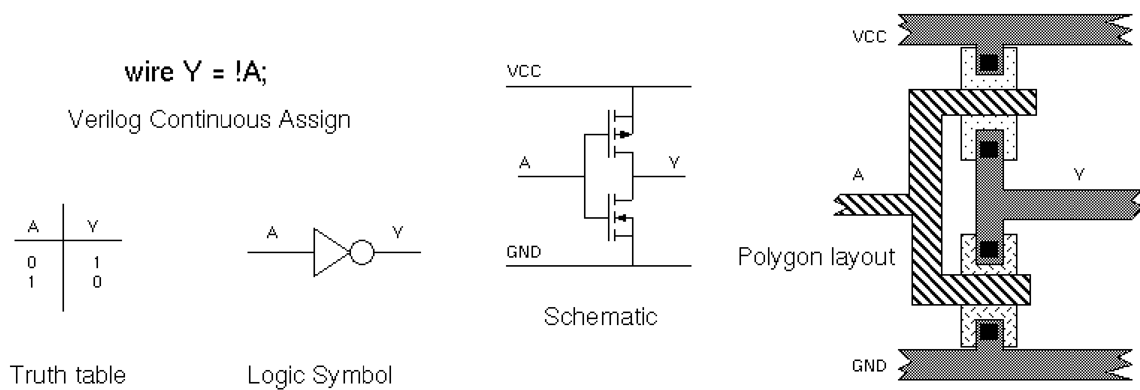


Figure 7: An inverter viewed at various levels of abstraction.

KG 1 — Energy use in Digital Hardware.

Battery life is very important for convenience. The electricity bill is sometimes the biggest cost in a data centre [citation needed!]. Saving energy in computing is always a good idea. In this section we will examine energy and performance and energy saving techniques.

Energy in an electronic device gets used in several different ways: for a Mobile Phone we might see the following budget:

- **Screen Backlight**: 1 to 2 watts
- **RF Transmisions**: via the various transmit antennae: up to 4 watts
- **Sound and Vibrations**: through the speaker and little shaker motor 300 mW
- **Heat**: 'wasted' in the electronics: up to 5 watts

A mobile phone might have a 10 Wh capacity (36 kilo Joules).

1.1 Basic Physics

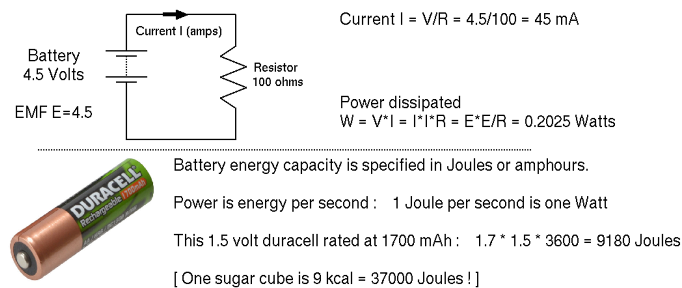


Figure 1.1: Ohms Law, Power Law and Battery Capacity.

We need to know that power (watts) is voltage (volts) times current (amps). Power (watts) is also energy (joules) per unit time (second).

And a joule of energy is one colomb of charge dropping one volt in potential.

$P = V \times I = E \times f$

1.1.1 Energy of Computation (1)

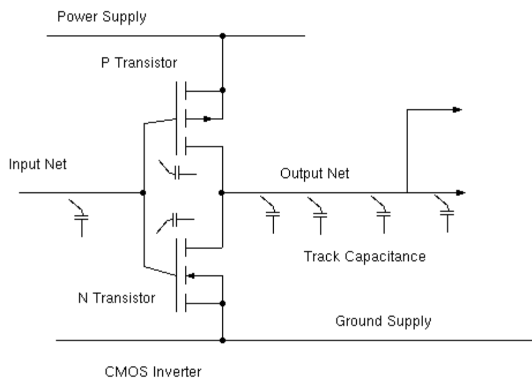
It is critical to understand where the energy goes in hardware in physical terms. All our models build up from this basis. Dynamic energy arising from stray capacitance turns out to be our main energy user today.

Gate current I = Static Current (leakage) + Dynamic Current.

Dynamic current = Short-circuit current + Dynamic charge current.

Early CMOS (VCC 5 volts): negligible static current, but today at VCC of 0.9 to 1.3 volts it can be 30 percent of consumption for some (high-leakage) designs.

The short-circuit power can generally be neglected. It is the energy wasted when both the P and N transistors are briefly conducting at once as the output. It is only significant when an input is left floating or slowly changing.



Dynamic charge current computation:

- All energy in a net is wasted each time output goes from one to zero.
- The energy in a capacitor was $E = CV^2/2$.
- The same amount of energy is wasted charging it up.
- Dominant capacitance is proportional to net length.
- Gate input and output capacitance also contribute to C.

The **toggle rate** for a net is the average frequency of a change in value, which is twice the activity ratio multipled by the clock frequency.

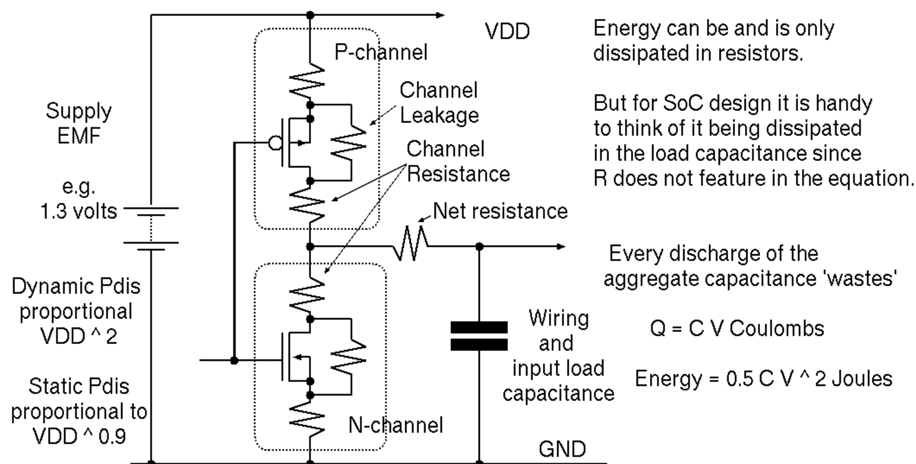


Figure 1.2: Dynamic energy dissipation mechanism.

Capacitors do not consume energy - they only store it temporarily. Only resistors dissipate energy in logic circuits, but their resistance does not feature in the energy use formula. The energy in the wiring capacitance is 'wasted' on each complete transistion cycle. If the clock frequency is f and a net has activity ratio α (the fraction of clock cycles it transitions from one to zero) then the energy used is

$$E = f * \alpha * C * V^2$$

As we increase voltage, dynamic power grows quadraticly. Static power grows a little better than (i.e. less than) linear since transistors may turn off more fully.

Note that although we divide by two as part of the standard formula for the energy in a capacitor, this quantity of energy is wasted both in the charging network on the zero-to-one transition and in the discharging network on the one-to-zero transition. So we can drop the half.

Note: static power consumption is static current multiplied by supply voltage (P=IV). Page 30 or so of this cell library has real-word examples: 90nm Cell LibrarySee also the power formula on the 7400A data

sheet: 74LVC00A.pdfFurther details: Power Management in CPU Design.

1.1.2 Landauer Limit and Reversible Computation

There are theoretical limits on the energy an irreversible computation requires. Current technology is a long way off these in two resepects:

- we use too much energy representing and communicating bits and
- we use Vonn Neumann based computation which does not scale well.

Let's consider electrical computers:

- If we build a computer using a network of standard components (such as transistors), where the inteconnection pattern expresses our design intent, then the components must be at different spatial locations. The computer will have some physical volume.
- If we interconnect our components using electrical wires these will have capacitance, resistance and inductance that stop them behaving like ideal conductors. The smaller the volume, the less wire we need and the better the wires (and hence computer) will work.
- If we use transistors that need a swing of about 0.7 volts on their gates to switch them reliably between off and on, then our wires need to move through at least that much potential difference.

The capacitance of the wires turns out to be our main enemy. Given a prescribed minimum voltage swing, the energy used by switching a wire between logic levels can only be made smaller by reducing its area and hence capacitance. Hence smaller computers are better.

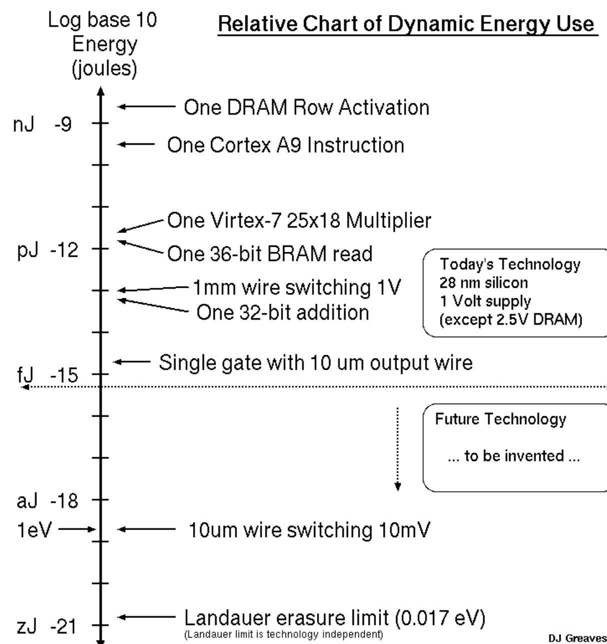


Figure 1.3: Relative chart of dynamic energy use.

The traditional approach of wasting the energy of each transition can be countered by returning the charge to the power supply (eg. Asynchrobatic logic for low-power VLSI design by David John Willingham, Univ Westminster, 2010) and reversible logic (eg. Toffoli logic) can get below the Landauer limit. Some standard algorithms such as encryption and lossless compression are reversible at the large - the trick is

mooted to be to code in a way that does not delete intermediate results during the computation. Such techniques may be in wide use within a decade or perhaps two.

Figure 1.3 shows ballpark figures for dynamic energy use in today's (2016/2018) mainstream 28 nm silicon technology. We see that contemporary computers are about six orders of magnitude above the Landauer limit in terms of energy efficiency, so there is a lot of improvement still possible before we have to consider reversibility. (Log to base 18 months of one million is roughly 30 years, so 2050 is the reversible computing deadline.)

Note: Reversible computing not examinable for Part II.

1.1.3 Typical Numerical Values

Let's just run some numbers for a fictional chip made solidly of logic:

22 um geometry: Capacitance per micron 0.3 fF

Energy of discharge for a net of 1mm at VDD of 1 volt is 0.15 pJ

Simple gate area: 0.3 square um.

If we assume a sub-system of 1000 gates, its area is 1000 x 0.3 = 300 sq um

By Rent or otherwise, we have an average wiring length of 0.3 x sqrt(300) = 5.2 um.

Clocking at 200 MHz with an activity ratio of 0.1, dynamic power will be 1000 x 200e6 x 0.1 x 5.2 x 0.3e-15 = 31 uW.

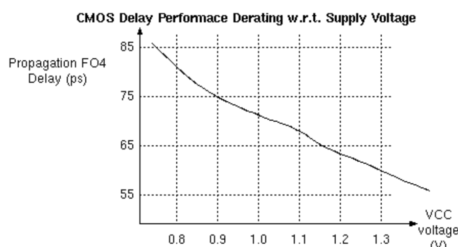
We will assume the inputs transition quickly enough for us to neglect short-circuit currents. Now we must add on the static power and the energy to drive the outputs from the sub-system:

Static power depends on leakage, but might be one fifth the dynamic power at 1 volt, although depends greatly on activity ratios and choice and semiconductor doping levels.

If we assume ten output nets with an activity of 0.1 and length 250 micron, their energy will be 10 x 200e6 x 0.1 x 250 x 0.3e-15 = 150 uW.

If we assume a 1 sq centimetre chip is made up of 1e8/300 = 3e5 such units, the chip power would be 3e5(150e-6 + 31e-6 + 6e-6)=56 watts. Clearly, we either need fluid cooling (water or ethanol heat pipe) or else the Dark Silicon approach. The activity ratio of 0.1 (toggle rate of 20 percent) reflects very-busy logic, such as an AES encoder. Most subsystems have lower activity ratios when in use. And the average activity ratio depends on how frequently used the block is, of course.

1.1.4 Gate Delay as a Function of Supply Voltage



Transistors have a gate threshold voltage around which they switch from off to on. This limits our lowest possible supply voltage. Above this, logic delay in CMOS is roughly inversely proportional to supply voltage. Accordingly, to operate faster, we need a higher supply voltage for a given load capacitance.

CMOS Delay Versus Supply Voltage

$$\text{Gate Delay} \propto \frac{C * V}{(V - V_t)^2}$$

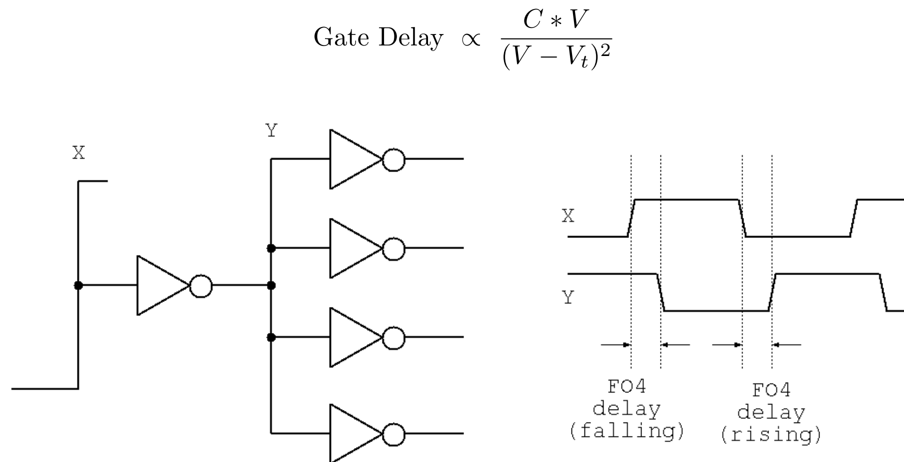


Figure 1.4: Fanout-4 delay measurement.

The **FO4 delay** is the delay through an inverter that is feeding four other nearby inverters (fan out of four). This is often quoted as a reference characteristic of a logic technology. The combinational delay of a particular design can also be expressed in a technology-independent way by quoting it in units of FO4 delay.

Here is a complete demo of simulating a CMOS inverter made from two MOSFETs using hspice. *Note: Spice simulation is not examinable for Part II CST.*

```
// spice-cmos-inverter-djg-demo.hsp
// Updated 2017 by DJ Greaves
// Based on demo by David Harris harrisd@leland.stanford.edu


///////////////////////////////////////////
// Set supply voltage



////////////////////////////////////////
// Declare global supply nets and connect them to a constant-voltage supply
.global Vdd Gnd
Vsupply Vdd Gnd DC 'VddVoltage'


///////////////////////////////////////////
// Set up the transistor geometry by defining lambda

.opt scale=0.35u  * Define lambda // This is half the minimum channel length.

// Set up some typical MOSFET parameters.
//http://www.seas.upenn.edu/~jan/spice/spice.models.html#mosis1.2um

.MODEL CMOSN NMOS LEVEL=3 PHI=0.600000 TOX=2.1200E-08 XJ=0.200000U
    +TPG=1 VTO=0.7860 DELTA=6.9670E-01 LD=1.6470E-07 KP=9.6379E-05
    +UO=591.7 THETA=8.1220E-02 RSH=8.5450E+01 GAMMA=0.5863
    +NSUB=2.7470E+16 NFS=1.98E+12 VMAX=1.7330E+05 ETA=4.3680E-02
    +KAPPA=1.3960E-01 CGDO=4.0241E-10 CGSO=4.0241E-10
    +CGBO=3.6144E-10 CJ=3.8541E-04 MJ=1.1854 CJSW=1.3940E-10
    +MJSW=0.125195 PB=0.800000


.MODEL CMOSP PMOS LEVEL=3 PHI=0.600000 TOX=2.1200E-08 XJ=0.200000U
    +TPG=-1 VTO=-0.9056 DELTA=1.5200E+00 LD=2.2000E-08 KP=2.9352E-05
    +UO=180.2 THETA=1.2480E-01 RSH=1.0470E+02 GAMMA=0.4863
    +NSUB=1.8900E+16 NFS=3.46E+12 VMAX=3.7320E+05 ETA=1.6410E-01
    +KAPPA=9.6940E+00 CGDO=5.3752E-11 CGSO=5.3752E-11
    +CGBO=3.3650E-10 CJ=4.8447E-04 MJ=0.5027 CJSW=1.6457E-10
    +MJSW=0.217168 PB=0.850000
```

```
/////////////////////////////////////////////
// Define the inverter, made of two mosfets as usual, using a subcircuit.

.subckt myinv In Out N=8 P=16 // Assumes 5 lambda of diffusion on the source/drain
m1 Out In Gnd Gnd CMOSN l=2 w=N
+ as='5*N' ad='5*N'
+ ps='N+10' pd='N+10'
m2 Out In Vdd Vdd CMOSP l=2 w=P
+ as='5*P' ad='5*P'
+ ps='P+10' pd='P+10'
.ends myinv


//////////////////////////////////////////////
// Top-level simulation netlist
//  One instance of my inverter and a load capacitor
x1 In Out  myinv         // Inverter
C1 Out Gnd 0.1pF         // Load capacitor

//////////////////////////////////////////////
// Stimulus: Create a waveform generator to drive In
// Use a  "Piecewise linear source"  PWL that takes a list of time/voltage pairs.

Vstim In Gnd PWL(0 0 1ns 0   1.05ns 'VddVoltage'   3ns VddVoltage    3.2ns 0)

//////////////////////////////////////////////
// Invoke transient simulation (that itself will first find a steady state)

.tran .01ns 6ns // Set the time step and total duration
.plot TRAN v(In) v(Out)
.end
// To get an X-windows plot, run the following interactive sequence of commands:
// $ ngspice spice-cmos-inverter-djg-demo.hsp
// ngspice 11 -> run
// ngspice 12 -> plot v(In) v(Out)

// end of file
```

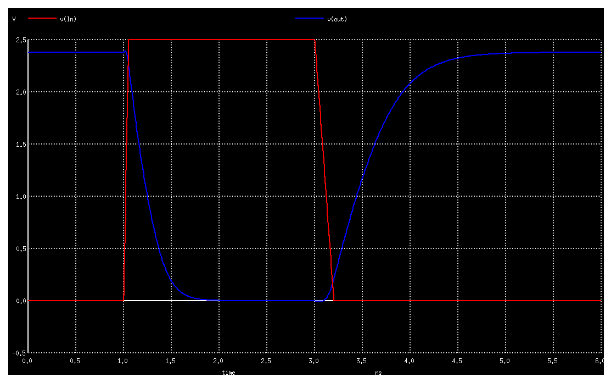


Figure 1.5: Plot when running from a VCC supply of 2.5 volts. Red is stimulus and blue is output.

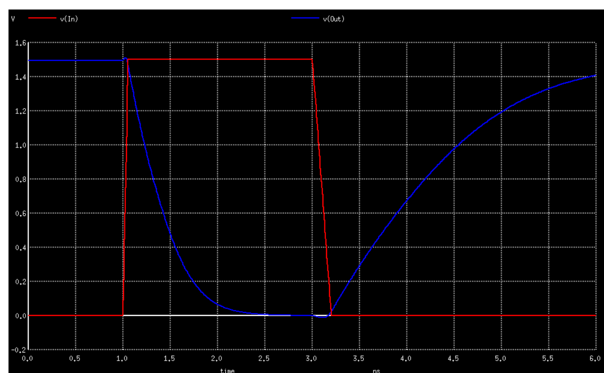


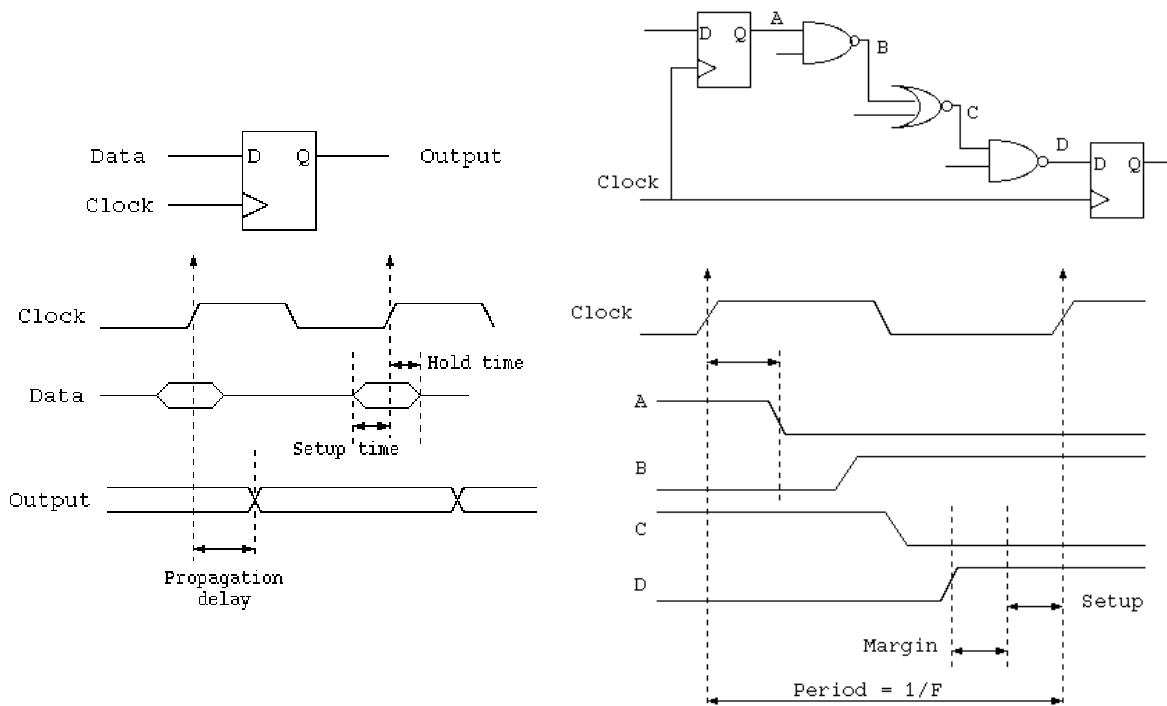
Figure 1.6: Plot when running at 1.5 volts.

The output load capacitor experiences a fairly typical exponential charge and discharge shape. The shape

is not a true 1-exp(-t/CR) curve owing to some non-leariy in the MOSFETs. But it is pretty close. If the FETs had equal on resistance at the two supply voltages, although the swing of the ouput in the two plots would be different, the delays before they cross the half-supply level would be identical. The difference arises owing to the on resistance being less when the gate voltage is less (i.e. when it is closer to the transistor threshold voltage).

1.1.5 Detailed Delay Model.

Now we have looked at the energy model, the other primary metric for a design can be considered: its delay performance that limits the clock rate.



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

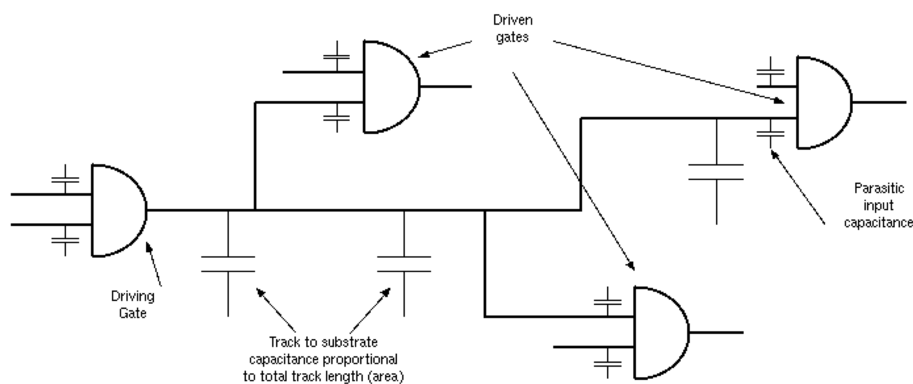


Figure 1.7: Logic net with tracking and input load capacitances illustrated.

Both the power consumption and effective delay of a gate driving a net depend mainly on the length of the net driven.

The delay is modelled with this formula:

$$\text{device delay} = (\text{intrinsic delay}) + (\text{output load} \times \text{derating factor}).$$

The track-dependent output loading is a library constant times the track area. The load-dependent part is the sum of the input loads of all of the devices being fed. For short, non-clock nets (less than 0.1 wavelength), we just include propagation delay in the gate derating and assume the signal arrives at all points simultaneously.

Precise track lengths are only known after place and routing (Figure 6). Pre-layout and pre-synthesis we can predict net lengths from Rent's Rule and RTL-level heuristics.

Figure 1.7 shows a typical net, driven by a single source. To change the voltage on the net, the source must overcome the stray capacitance and input loads. The fanout of a gate is the number of devices that its output feeds. The term *fanout* is also sometimes used for the maximum number of inputs to other gates a given gate is allowed to feed, and forms part of the design rules for the technology.

The speed of the output stage of a gate, in terms of its propagation delay, decreases with output load. Normally, the dominant aspect of output load is capacitance, and this is the sum of:

- the capacitance proportional to the area of the output conductor,
- the sum of the input capacitances of the devices fed.

To estimate the delay from the input to a gate, through the internal electronics of a gate, through its output structure and down the conductor to the input of the next gate, we must add three things:

- the internal delay of the gate, termed the intrinsic delay
- the reduction in speed of the output stage, owing to the fanout/loading, termed the derating delay,
- the propagation delay down the conductor.

The propagation delay down a conductor obeys standard transmission line formula and depends on the distributed capacitance, inductance and resistance of the conductor material and adjacent insulators. For circuit board traces, resistance can be neglected and the delay is just the *speed of light* in the circuit board material: about 7 inches per nanosecond, or 200 metres per microsecond. On the other hand, for shorter nets on chip, less than one tenth a wavelength long, we commonly assume the signal arrives at all destinations at once and model the propagation delay as an additional inertial component of the driving gate and include this via the gate derating.

1.2 Memory Power and Performance

As well as logic gates, our main users of power will be SRAM, DRAM and I/O pads.

1.2.1 RAM - On-chip Static Random-Access Memory (Static RAM).

RAMs vary in their size and number of ports. Single-ported SRAM is the most important and most simple resource. It connects to a bus as an addressable target. It is also used inside caches for tags and data. Today's SoC designs have more than fifty percent of their silicon area devoted to SRAM for various purposes. Commonly, synchronous RAMs are used, requiring typically one clock cycle of delay between address input and corresponding data output. The same address can be written with fresh data during the same clock cycle, if desired.

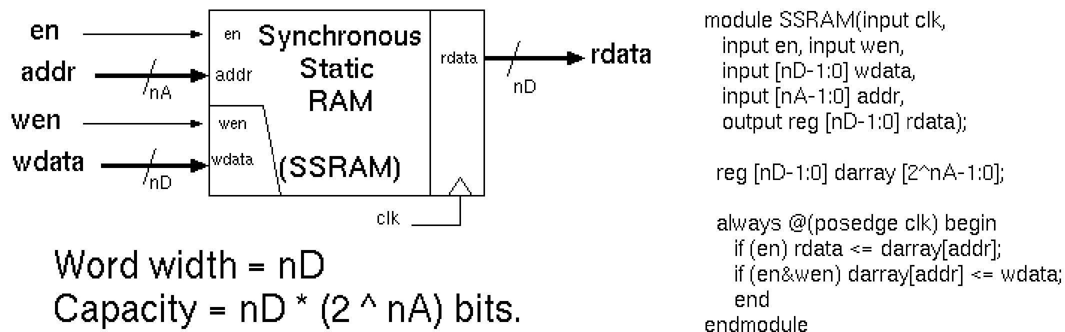


Figure 1.8: Synchronous static RAM with single port: logic symbol and internal RTL model.

The illustrated RAM has a one clock cycle read latency. When a write occurs, the old value at the location is still read out, which is commonly a useful feature.

The 'en' input signal is not striclty needed since the RAM could deliver read data on all cycles. However, this wastes power, so without an enable input we should ensure the address inputs are relatively stable when a RAM result is not needed.

Owing to RAM fabrication overheads, RAMs below a few hundred bits should typically be implemented as register files made of flip-flops. But larger RAMs have better density and power consumption than arrays of flip-flops.

RAMs for SoCs were originally supplied by specialist companies such as Virage and Artizan (although these are now part of larger EDA companies). A 'RAM compiler' tool is run for each RAM in the SoC. It reads in the user's size, shape, access time and port definitions and creates a suite of models, including the physical data to be sent to the foundry.

High-density RAM (e.g. for L2 caches) may clock at half the main system clock rate and/or might need error correction logic to meet the system-wide reliability goal.

RAM consumes static and dynamic energy. The ESL section of these notes gives high-level modelling figures that include about 10 pJ per read or write operation and a leakage of 82 nW per bit.

Additional notes:

On-chip SRAM needs a test mechanism. Various approaches:

- Can test with software running on embedded processor.
- Can have a special test mode, where address and data lines become directly controllable (JTAG or otherwise).
- Can use a built-in hardware self test (BIST) wrapper that implements 0/F/5/A and walking ones typical tests.

Larger memories and specialised memories are normally off-chip for various reasons:

- Large area: would not be cost-effective on-chip,
- Specialised: proprietary or dense VLSI technology cannot best be made on a SoC die where the process is optimised for general logic,
- Specialised: e.g. non-volatile process (such as FLASH)
- Commodity parts: economies of scale (ZBT SRAM, DRAM, FLASH)

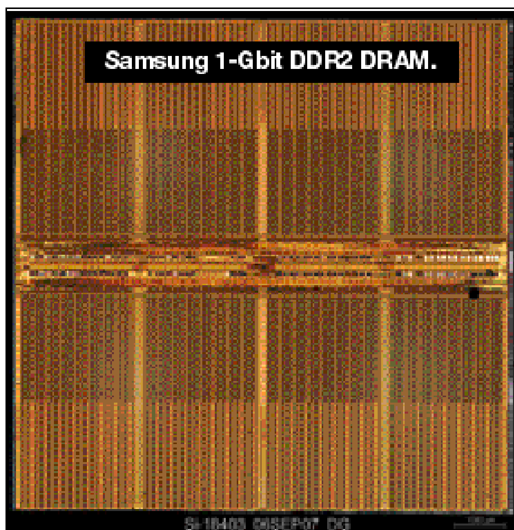
But in the last five years DRAM and FLASH have found their way onto the main SoC as maturing technology shifts the economic sweet spot.

1.2.2 Dynamic RAM : DRAM

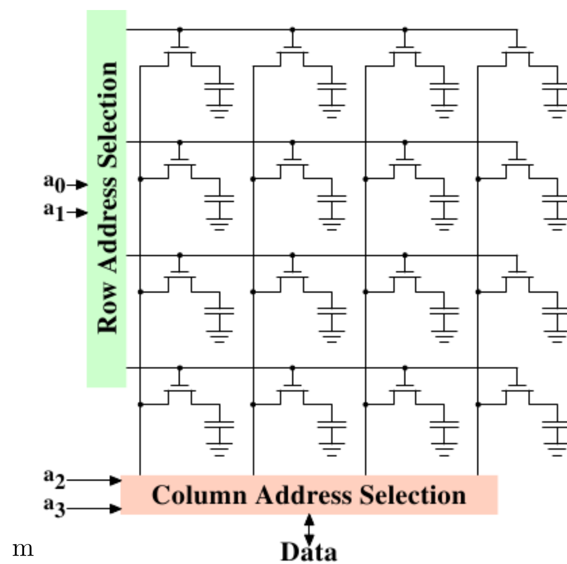
Figure 1.9: DRAM single-in-line memory module (SIMM).

DRAMs for use in PCs are mounted on SIMMS or DIMMS. But for embedded applications, they are typically just soldered to the main PCB. Normally one DRAM chip (or pair of chips to make D=32) is shared over many sub-systems in, say, a mobile phone. SoC DRAM compatibility might be a generation behind workstation DRAM: e.g. using DDR3 instead of DDR4 Also, the most recent SoCs embed some DRAM on the main die or flip-chip/die-stack the DRAM directly on top of the SoC die in the same package (multi-chip module — MCM).

Modern DRAM chip with 8 internal memory banks.



m



These are the pin connections of a typical DIMM from 2010:

Clk+/-	Clock (200MHz)
Ras-	Row address strobe
Cas-	Column address strobe
We-	Write enable
dq[63:0]	Data in/out
reset	Power on reset

wq[7:0]	Write lane qualifiers
ds[7:0]	Data strobes
dm[7:0]	Data masks
cs-	Chip select
addr[15:0]	Address input
bs[2:0]	Bank select
spd[3:0]	Serial presence detect

DRAM performance is often quoted in MT/s which is mega-transfers per second. Our DIMM example has a 200 MHz clock and hence 400 MT/s. This is low performance by today's standards: 64 bits times 400 MHz gives 25.6 Gb/s peak (4 GB/sec). The capacity is a 1 Gbyte DIMM made of 8 chips.

The latest (Jan 2018) DDR4 memories operate at 4332 MT/sec. Each transfer caries a word the width of the DRAM data bus (e.g. 16 bits) and transfers are performed on both edges of a clock. This clock would be at 2.166 GHz. But that is the burst transfer rate. To manage a burst, DRAM timings of 19-21-21 are used, which is the number of clock cycles to send the row address, the column address and for writeback, respectively.

In the worst case, if the DRAM is currently 'open' on the wrong row, 61 clock cycles will be needed to change to the new location. Roughly the same number of clock cycles again will be used in pipeline stages through the various memory hierarchy levels of the controlling device.

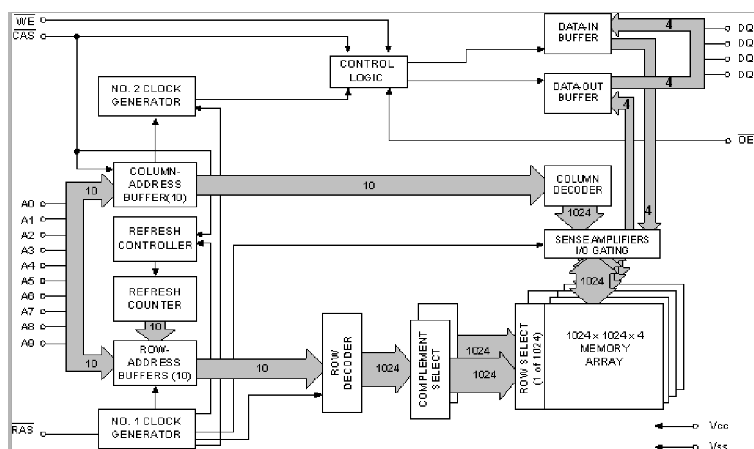


Figure 1.10: Single-bank, 4-bit wide, DRAM Chip Internal Block Diagram.

This DRAM has four data I/O pins and four internal planes, so no bank select bits. (Modern, larger capacity DRAMs have multiple such structures on their die and hence additional bank select inputs select which one is addressed.)

Dynamic RAM keeps data in capacitors. The data will stay there reliably for up to four milliseconds and hence every location must be read out and written back (refreshed) within this period. The data does not need to leave the chip for refresh, just transferred to the edge of its array and then written back again. Hence a whole row of each array is refreshed as a single operation.

DRAM is not normally put on the main SoC chip(s) owing to its specialist manufacturing steps and large area needs. Instead a standard part is put down and wired up. (DRAM is traded as a commodity like corn and gold.)

A row address is first sent to a bank in the DRAM. This is called opening the row or a row activation. One a row is open, one has random access to the columns of that row using different column addresses. The DRAM cells internally have destructive read out because the capacitors get discharged into the row wires when accessed. Therefore, whenever finished with a row, the bank containing it goes busy while it

- Die Size Efficiency
  - DRAM is as much as 6x smaller in comparison to SRAM on a per bit basis.
  - Cell size for SRAM roughly 6x the size of a DRAM cell
    - 6 transistor cell versus 1 transitor
  - For DRAM approximately 55% - 70% of the die size is array
  - Periphery circuitry is 45% to 50% larger for SRAM
    - Non multiplexed addressing

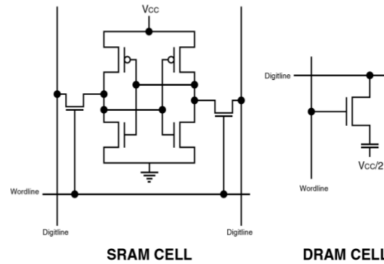


Figure 1.11: SRAM cell complexity versus DRAM cell

writes back the data and gets ready for the next operation (charing row wires to mid-way voltage etc.).

DRAM is slow to access and certainly not ‘random access’ compared with on-chip RAM. A modern PC might take 100 to 300 clock cycles to access a random part of DRAM, but the ratio may not be as severe in embedded systems with lower system clocks. Nonetheless, we typically put a cache on the SoC as part of the memory controller. The controller may embody error detection or correction logic using additional bit lanes in the DRAM.

The cache will access the DRAM in localised bursts, saving or filling a cache line, and hence we arrange for cache lines to lie within DRAM rows.

The controller may keep multiple banks open at once to exploit tempro-spatial access locality.

DRAM controller is typically coupled with a cache or at least a write buffer.

DRAM: high latency and write-back overheads imply me must select a bank closing policy. The best controllers will lookahead in a pool of pending requests to assist decisions on when to do write back (aka close or deactivate). It is normal to prioritise reads over writes, but overtaking must be avoided or else reads can be served from the write queue. But a new request falling in a previously open line can arrive just after we closed it! It is best if clients can tolerate responses out-of-order (hence use bus/NoC structure that supports this).

Controller must

- set up DRAM control register programming,
- set clock frequency and calibrate delay lines,
- implement specific RAS-to-CAS latencies and many other timing details,
- and ensure refresh happens.

Controller often contains a tiny CPU to interrogate serial device data. DRAM refresh overhead has minimal impact on bus throughput. For example, if 512 refresh cycles are needed in 4 ms and the cycle rate is 200E6 the overhead is 0.1 percent.

Another design consideration is how the system address bus is mapped to the various row, bank and column physical bits. This will alter the effect of memory layout on performance. Hardware is normally programmable in terms of physical address interleave. An example is, starting with most significant in physical address space: bank, row, col, burst offset, byte lane. Byte lane and burst offset are always at the bottom and col is kept lower than row, but having bank lower than col gives interleaving of accesses to open pages which is sensible when the system workload has a lot of localised activity to one large area, whereas having bank higher makes sense when the system has various concurrent active hot spots, such as typical with heap, stack, code and static segments. etc..

Here we are using the definition of a bank as a region where only one row is active within one bank. Multiple banks may be within the same channel (which always occurs for banks within one DRAM chip) or arranged over separate channels were a channel is defined as a data bus (or DRAM controller backside port). Multiple channles gives higher data throughput owing to spatial diversity.

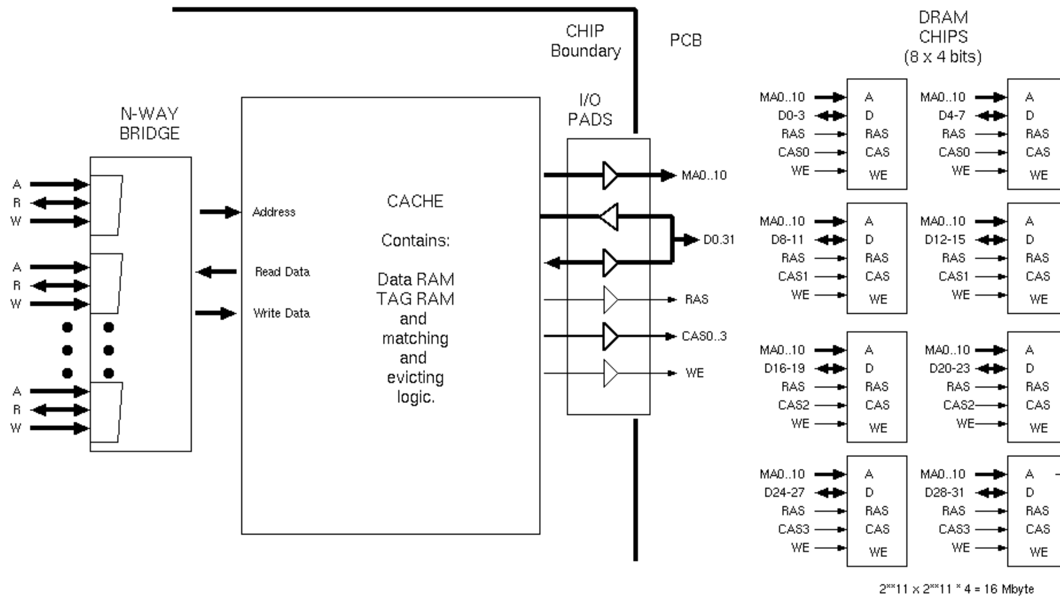


Figure 1.12: Typical structure of a small DRAM subsystem.

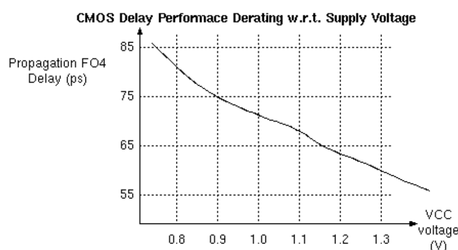
Figure 1.12 shows a 32-bit DRAM subsystem. Four CAS wires are used so that writes to individual byte lanes are possible. For large DRAM arrays, need also to use multiple RAS lines to save power by not sending RAS to un-needed destinations. [*Detailed pinouts and wiring details non-examinable*]

We will discuss DRAM energy use in the ESL section of the course. A main energy use is static power in the high-speed PCB driving and receiving pads. Each row activation and column selection takes dynamic energy.

1.3 The Voltage and Frequency Relationship

Looking at the derating graph for the standard cell libraries, we see that in the operating region, the frequency/voltage curve is roughly linear. CMOS delay is inversely proportional to supply voltage. A technique that exploits such curves is DVFS – dynamic voltage with frequency scaling.

Logic with higher-speed capabilities is smaller which means it consumes greater leakage current which is being wasted while we are halted. Also leakage energy is proportional to supply voltage (or perhaps sublinear with exponent 0.9ish : as we raise voltage, transistors are indeed turned off more, but P=IV is still increasing).



If we vary the voltage to a region dynamically, while keeping f constant, a higher supply voltage uses more power (square law) but would allow a higher f.
Let's only raise VCC when we ramp up f: classical DVFS.

For a fixed task size, energy use is proportional to V squared, so DVFS is the ideal method (i.e. for predictable, real-time tasks in low-leakage technology):

1. Adjust clock f for just-in-time completion (e.g. in time to decode the next frame of a video),
2. then adjust VCC to minimal value for reliably meeting the set-up time.

In general servers (ie. not for a static/finite workload), ramping voltage up linearly with clock frequency (f) results in dynamic power consumption with a cubic dependence on f. But work may be bursty, so DVFS is applied (e.g. by a laptop governor).

DVFS obtains peak performance under heavy loads, yet avoid cubic overhead when idle. We adjust VCC so that, at all times, the logic just works. However, we need to keep close track of whether we are meeting real-time and timing closure deadlines.

In a server farm processing blade we may be thermally limited, so DVFS will be throttled back by rack-level governors or Intel's RAPL.

Minor caveats (non-examinable):

Additional notes:

Combinational logic cannot be clock gated (e.g. PAL and PLA). For large combinational blocks: can dip power supply to reduce static current when block is completely idle (detect with XORs).

So a typical SoC uses not only many dynamic clock gated islands, but also some sub-continents with automatic frequency and voltage variation. Power isolation originally used on a longer and larger scale (complete continents) but now a lot of power islands are being used.

It is possible to locally and quickly adjust supply voltage with a series transistor - but wasteful compared with an off-chip switched-mode regulator.

An off-chip power supply can be efficiently adjusted, but limited to only a few voltage islands and tens of milliseconds inertia.

1.3.1 DVFS Example

Example: core area 64 mm^2; average net length 0.1 mm; 400K gates/mm^2, $a = 0.25$.

Net capacitance = 0.1 mm × 1 fF/mm × 400K × 64 mm^2 = 2.5 nF.

Supply Voltage (V)	Clock Freq (MHz)	Static Power (mW)	Dynamic Power (mW)	Total Power (mW)
0.8	100	40	24	64
1.35	100	67	68	135
1.35	200	67	136	204
1.8	100	90	121	211
1.8	200	90	243	333
1.8	400	90	486	576

The table shows example power consumption for a circuit when clocked at different frequencies and voltages. The important thing to ensure is that the supply voltage must be sufficient for the clock frequency in use: too low a voltage means that signals do not arrive at D-type inputs in time to meet set up times.

Power consumption versus frequency is worse than linear: it goes with a power law.

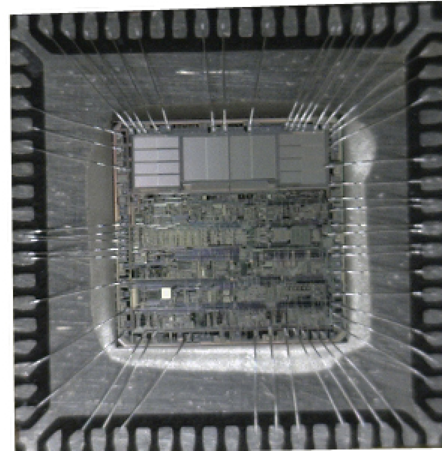
In the past, chips were often **core-bound** or **pad-bound**. Pad-bound meant that the chip had too many I/O signals for its core logic area: the number of I/O's puts a lower bound on the perimeter of the chip. Today's VLSI technology allows I/O pads in the middle of the chip and designs are commonly

power-bound.

1.3.2 90 Nanometer Gate Length.

The mainstream VLSI technology in the period 2004-2008 was 90 nm. This had low leakage and very high wafer yields. Now the industry is using 22 nanometer and smaller. Parameters from a 90 nanometer standard cell library:

Parameter	Value	Unit
Drawn Gate Length	0.08	μm
Metal Layers	6 to 9	layers
Max Gate Density	400K	gates/mm^2
Finest Track Width	0.25	μm
Finest Track Spacing	0.25	μm
Tracking Capacitance	1	fF/mm
Core Supply Voltage	0.9 to 1.4	V
FO4 Delay	51	ps
Leakage current		nA/gate



Typical processor core: 200k gates + 4 RAMs: one square millimeter. Typical SoC chip area is 50-100 mm^2 $\rightsquigarrow$ 20-40 million gates (semi-custom/standard cell). Actual gate and transistor counts are higher owing to full-custom blocks (RAMs mainly).

- 2007: Dual-core Intel Itanium2: 1.6 billion transistors (90 nm).
- 2010: 8-core Intel Nehalem: 2.3 billion transistors (45 nm).
- 2010: Altera Stratix IV FPGA: 2.5 billion transistors (40 nm).
- 2015: Intel CPU: circa 10 billion transistors (19 nm).
- 2018: ITRS predicts 7 nm technology for this year (2018)!

Moore's Law Transistor Count Dennard Scaling

Dimension Increase in Metal-Oxide-Semiconductor Memories and Transistors

Dennard's Rule stated that as transistors get smaller, their power density stays constant, so that the power use stays in proportion with area: both voltage and current scale (downward) with length. This meant that no new heat extraction technology was needed as VLSI capabilities improved. But once a supply voltage of 1 volt was reached, silicon CMOS cannot be run at much lower voltages without leakage (static power) greatly increasing.

Additional notes:

Typical modern datasheet rubric:

"The Xilinx Kintex UltraScaleTM FPGAs are available in -3, -2, -1, and -1L speed grades, with -3 having the highest performance. The -1L devices can operate at either of two VCCINT voltages, 0.95V and 0.90V and are screened for lower maximum static power. When operated at VCCINT = 0.95V, the speed specification of a -1L device is the same as the -1 speed grade. When operated at VCCINT = 0.90V, the -1L performance and static and dynamic power is reduced."

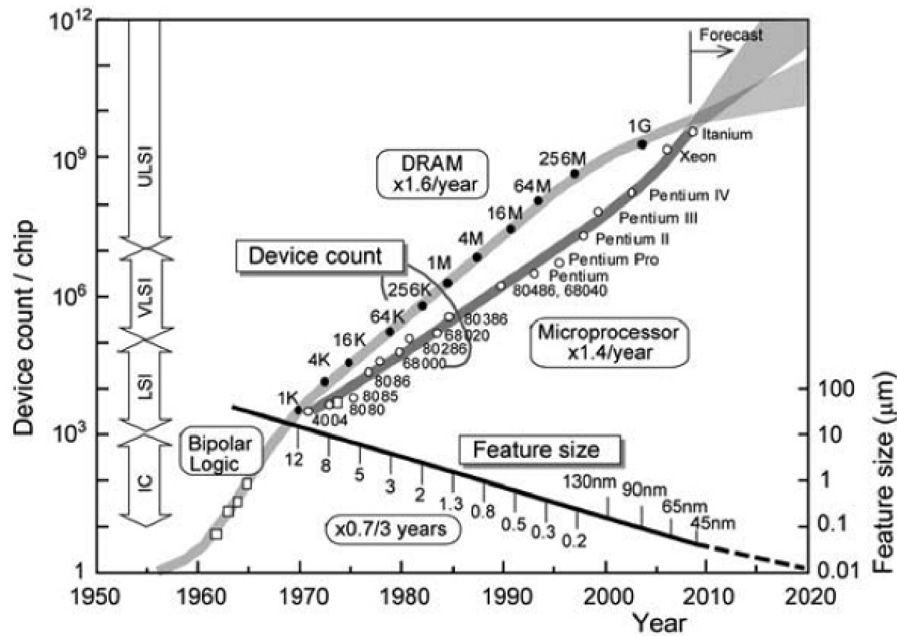


Figure 1.13: Technology Scaling Prediction.

The slide shows typical parameters from a 90 nanometer standard cell library. This figure refers to the width of the gate in the field effect transistors. The smaller this width, the faster the transistor can operate, but also it will consume more power as static leakage current. The 90 nm figure was the mainstream VLSI technology in the period 2004-2008, but then 40-45 nanometer technology was widely used with smaller 22 nm now mainstream.

Typical processor core: 200k gates + 4 RAMs: one square millimeter.

A typical SoC chip area is 50-100 mm^2 with 20-40 million gates. Actual gate and transistor count would be higher owing to custom blocks (RAMs mainly), that achieve a better denisty than standard cells.

Moore's Law has been tracked for the last three plus decades, but have we now reached the *Silicon End Point*? That is, can we no longer make things smaller (at the same cost)? Modern workstation processors have certainly demonstrated a departure from the previous trend of ever rising clock frequencies: instead they have several cores.

The **Power Wall** is currently the limiting factor for practical VLSI. As Horowitz points out, the fixed threshold voltage of transistors means that supply voltages cannot be reduced further as we go to smaller and smaller geometries, hence the previous technology trajectory will change direction: Scaling, Power, and the Future of CMOS. The limiting factor for commercial products has become the cost of thermal management. We can put more-and-more transistors on our chip but we cannot use them all at once - hence **Dark Silicon**.

1.4 Further Power Saving Techniques

Turning off and slowing down are our friends when it comes to saving power.

We can save power by controlling power supplies and clock frequencies: Figure 1.17.

Our first power saving technique uses DVFS. We clock quickly and halt or clock slowly and finish just in time. We aim to clock at the lowest suffcient voltage and clock frequency and with a as high as possible and minimal halt cycles.

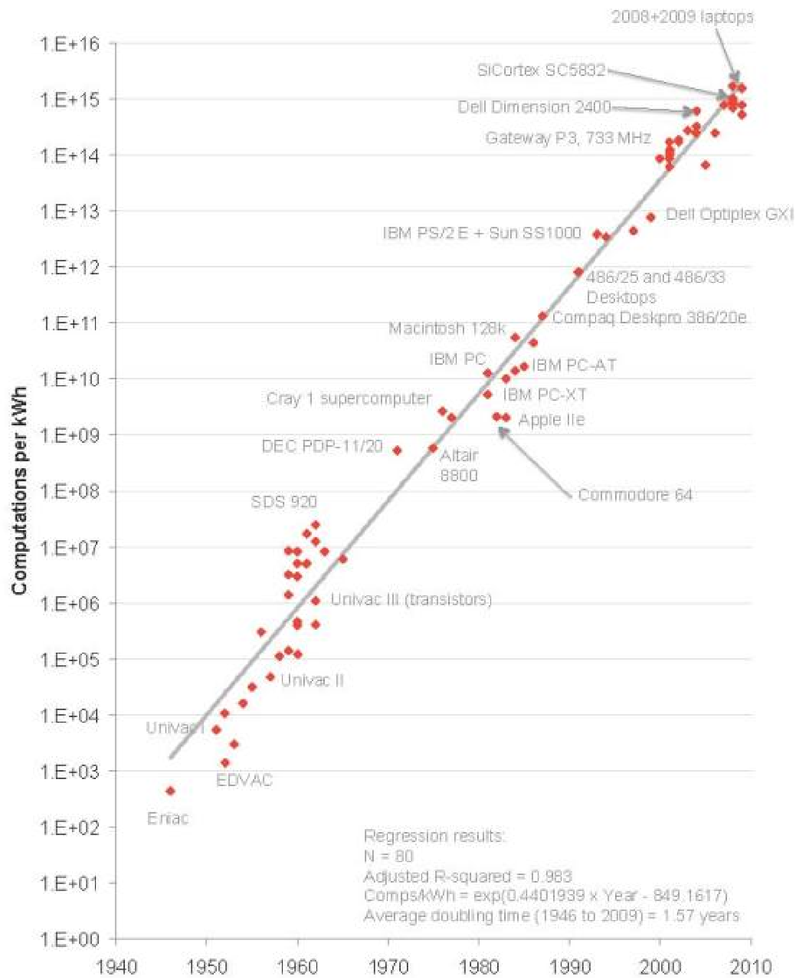


Figure 1.14: Computations per kWh. Divide by 3.6E6 for per Joule.

Frequency scaling means adjusting the clock frequency to a subsystem. The voltage will have to be scaled in tandem to ensure timing is met without wasting energy. Frequency adjustment can be instant if divider output is directly used. (But when PLLs with analog filters are used, there is inertia, e.g. 1 millisecond). Voltage adjustment also has inertia and there is design and implementation complexity supporting more than a few voltage regions.

	Clock Gating	**Supply Gating**	**DVFS**
Control:	automatic	various	software
Granularity:	register / FSM	larger blocks	macroscopic.
Clock Tree:	mostly free runs	turned off	slows down.
Response time:	instant	2 to 3 cycles	instant (or ms if PLL adjusted)
Proportionally vary voltage:	not possible	n/a	yes.

1.4.1 Save Power 2: Dynamic Clock Gating

Clock trees consume quite a lot of the power in an ASIC and considerable savings can be made by turning off the clocks to small regions. A region of logic is idle if all of the flip-flops are being loaded with their current contents, either through synchronous clock enables or just through the nature of the design. EDA

The following table shows the VDD power consumption of the E16G301 as a function of frequency and voltage with all 16 cores executing a heavy duty workload. The measurements were taken at room temperature without a heat sink and with 0 m/s airflow. The maximum operating frequency at each voltage level is specified next to the data point in the figure

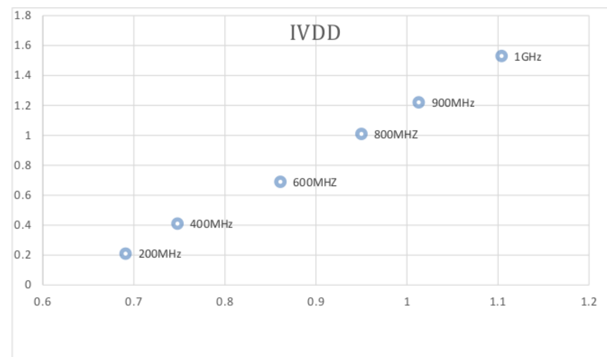


Figure 1.15: Epiphany 16 Core ‘Supercomputer’ Chip DVFS Points (a plot, not a table).

Figure 1.16: Thermal management heat pipes in a modern laptop.

DESIGNLINE

Instead of using synchronous clock enables, current design practice is to use a clock gating insertion tool that gates the clock instead. One clock control logic gate serves a number of neighbouring flip-flops: state machine or broadside register.

Problem with AND gate: if CEN changes when clock is high: causes a glitch. Problem with OR gate: if CEN changes when clock is low: causes a glitch. Hence, care must be taken not to generate glitches on the clock as it is gated. Transparent latches in the clock enable signal prevent these glitches.

Care needed to match clock skew when crossing to/from non-gated domain: avoid *shoot-through* by building out the non-gated parts as well. Shoot-through occurs when a D-type is supposed to register its current D input value, but this has already changed to its new value before the clock signal arrives.

How to generate clock enable conditions ? One could have software control for complete blocks (additional control register flags, as per power gating). But today’s designs automatically detect on a finer-grain basis. Synthesiser tools can automatically insert clock required conditions and insert the additional logic. Automatic tools compute ‘clock needed’ conditions. A clock is ‘needed’ if any register will change on a clock edge.

A lot of clock needed computation can get expensive, resulting in no net saving, but it can be effective if computed once at head of a pipeline.

	Clock	Power
On./Off	Clock gating	Power supply gating
Variable	Dynamic frequency scaling (DFS)	Dynamic voltage scaling (DVS)

Figure 1.17: Terminology and Overview of Power Saving Techniques.

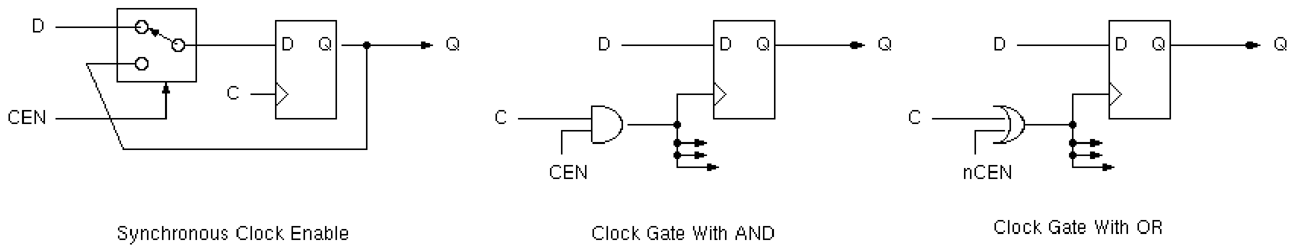


Figure 1.18: Clock enable using multiplexor, AND and OR gate.

If not a straightforward pipeline, need to be sure there are no 'oscillating' stages that retrigger themselves or an 'earlier' stage (add further runtime checks or else statically know their maximum settling time and use a counter). The maximum settling time, if it exists, is computed in terms of clock cycles using static analysis. Beyond the settling time, all registers will be being re-loaded with their current data on each clock cycle.

Beyond just turning off the clock or power to certain regions, we can consider further power saving techniques: dynamic frequency and voltage scaling.

1.4.2 Save Power 3: Dynamic Supply Gating

Increased tendency towards multi-product platform chips means large functional blocks on silicon may be off for complete product lifetime. The 'dark silicon' future scenario implies all chips must be mostly powered off. Battery powered devices will also use macro-scale block power down (e.g. the audio or video input and output subsystems).

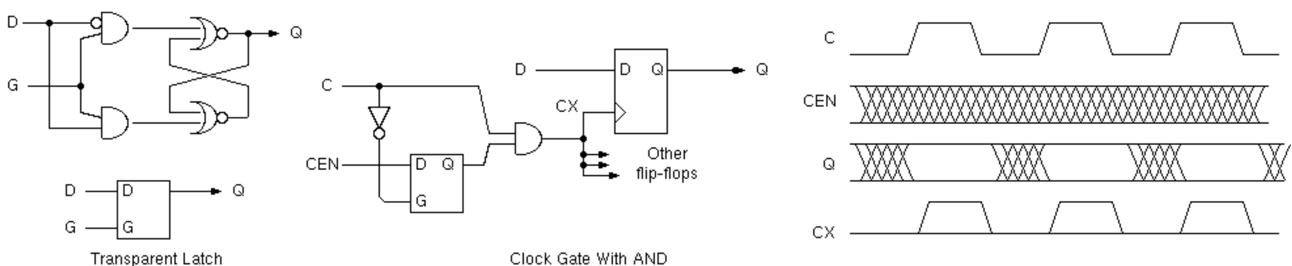


Figure 1.19: Illustrating a transparent latch and its use to suppress clock gating glitches.

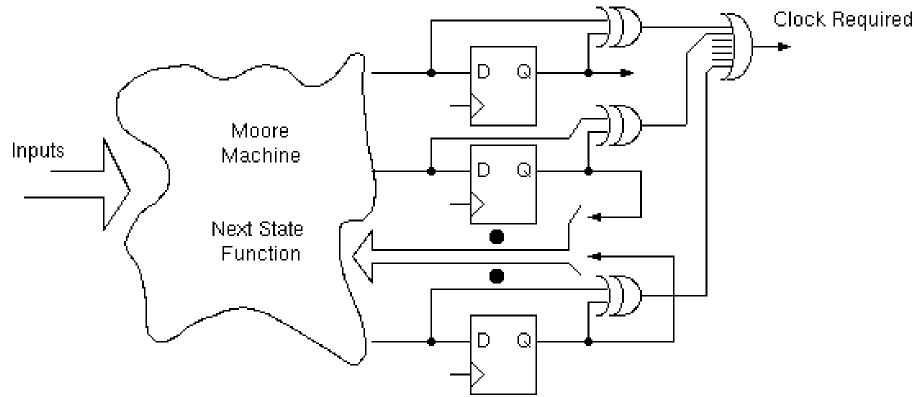


Figure 1.20: Using XOR gates to determine whether a clock edge would have any effect.

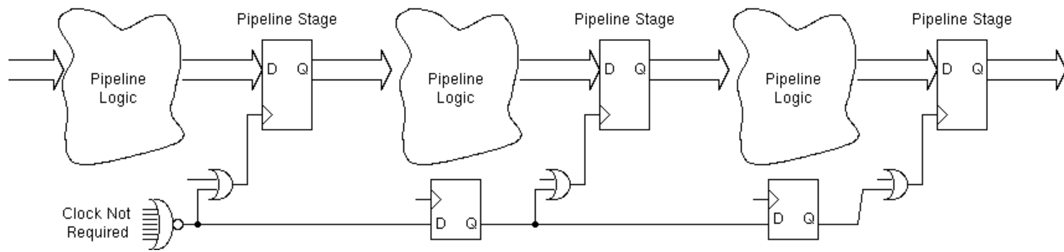
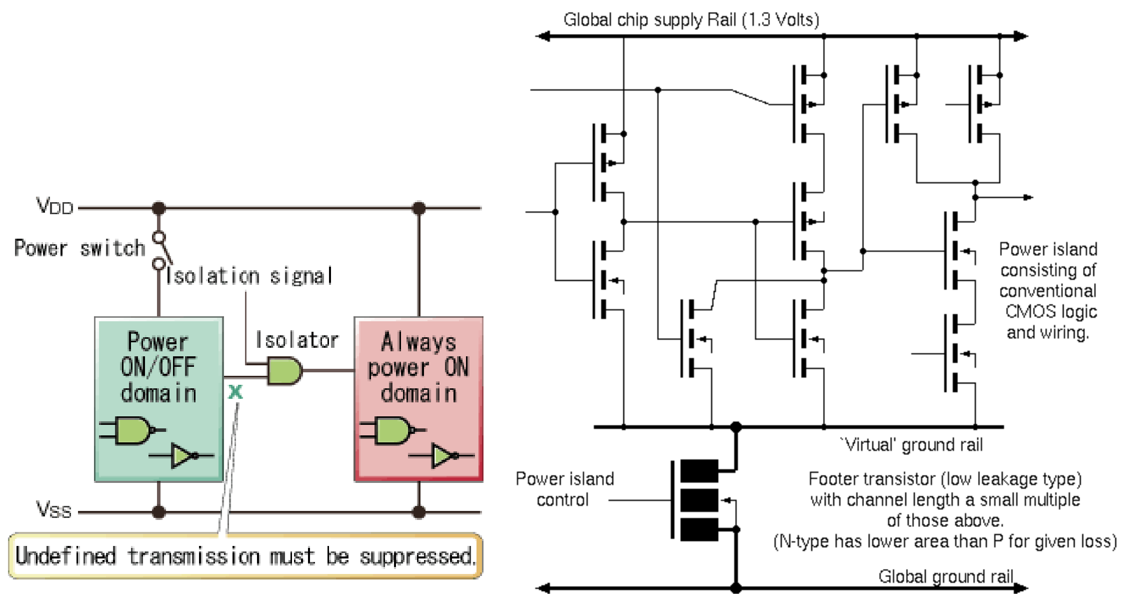


Figure 1.21: Clock needed computations forwarded down a pipeline.



Dynamic power gating techniques typically require some sequencing: several clock cycles to power up/down a region and enable/disable isolation gates.

Fujitsu Article: Design of low power consumption LSIs

Previously we looked at dynamic clock gating, but we can also turn off power supply to regions of a chip with fine or coarse gain, creating so-called power islands. We use power gating cells in series with supply rails. These are large, slow, low-leakage transistors. (Best to disconnect the ground supply since an N-channel transistor can be used which has smaller area for same resistance.)

Signal isolation and retention cells (t-latches) on nets that cross in and out of the region are needed. There is no register and RAM data retention in a block while the power is off. This technique is suitable at coarse grain for complete sub-systems of a chip that are not in use on a particular product or for quite a long time, such as a bluetooth tranceiver or audio input ADC. It can also be used on a fine grain with automated control similar to clock gating.

However, power gating requires some sequencing to activate the enables to the isolation cells in the correct order and hence several clock cycles or more are needed to power up/down a region. Additionally, gradual turn on over tens of milli-seconds avoids creating noise on the global power rails. Originally, power off/on was controlled by software or top-level input pads to the SoC. Today, dedicated microsequencer hardware might control a hundred power islands within a single subsystem.

A common practice is to power off a whole chip except for a one or two RAMs and register files. This was particularly common before FLASH memory was invented, when a small battery is/was used to retain contents using a lower supply (CMOS RAM data holding voltage). Today, most laptops, tablets and PCs have a second, tiny battery that maintains a small amount of running logic when the main power is off or battery removed. This runs the real-time clock (RTC).

Another technique that saves power is to half-turn-on a power gating transistor and thereby run an island at a lower voltage. This is not as efficient as adjusting standard switched-mode power supplies, since the half-turned on transistor will waste energy itself.

1.4.3 DVFS in Low-Leakage Technology

Considering DFVS again, there are (were?) two potential strategies (Aesop: Tortoise v Hare):

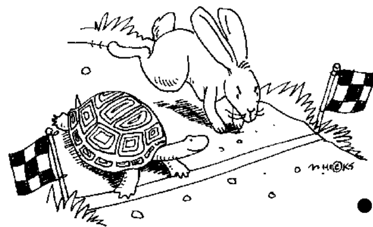


Figure 1.22: Aesop's Fable - Whose approach is better?

1. Compute quickly with halt(s), or
2. Compute more slowly and finish just in time.

To compute quickly and halt we need a higher frequency clock but consume the same number of active cycles. So the work-rate product, af, unchanged, so no power difference ? No. Running the same number of work cycles at a lower frequency requires a lower voltage and hence we save energy according to V^2.

But: current geometries only have a narrow operating voltage range Too low -> too much leakage. Too high -> tunnelling and wear and heat. So today, we operate at around one volt always and the trade off is just between high and low leakage technology - a static fab-time decision. There are further (unexaminable) prospects on the table, such as dynamic body bias ...

1.4.4 Static and Dynamic Power Tradeoff

But: for sub 45nm, voltage scaling is less viable and transistors with much higher static leakage current are commonly used: so can now be better to operate within the voltage/frequency band that works and then power off until next deadline.

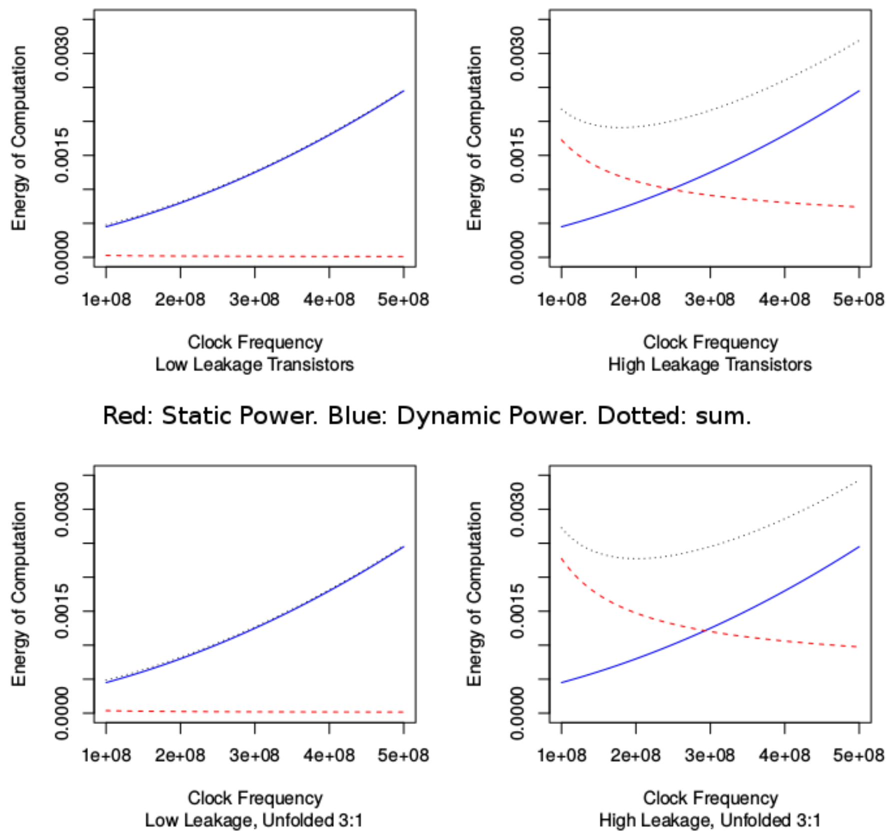


Figure 1.23: Sweetspot shift in DVFS approach for higher leakage on a real-time task.

```
# Trade off of Hare and Tortoise for increasingly leaky technology.
# For a hard-realtime computation we known the number of clock cycles needed but should we do them quickly
# and halt (Archilies) or slowly and finish just in time (Tortoise). For a higher leakage technology,
# switching off as much as possible tends to become preferable to running at low supply voltage.

# Unfold=1 is baseline design. Unfold=3 uses three times more silicon.
static_dynamic_tradeoff <- function(clock_freq, leakage, unfold, xx)
{
  op_count <- 1e5;
  execution_time = op_count / clock_freq / (unfold ^ 0.75);   // Model: Pollack-like unfold benefit.
  vdd <- 1 + 0.5 * (clock_freq/100e6);                        // Model: Higher supply needed for higher clk.
  static_power <- leakage * vdd ^ 0.9 * unfold * 0.4;         // Model: Leakage slightly sublinear.
  static_energy <- static_power * execution_time;             // Integrate static power
  dynamic_energy <- op_count * vdd ^ 2.0 / 0.5 * 1e-9;        // Use CV^2/2 for dynamic
}
```

For the 90nm technology, there was low static leakage and considerable scope for DVFS. With the smaller geometries performance can be traded off for greater leakage. Transistor dopant levels (and hence leakage) can be adjusted in regions or globally. We will want a low leakage, large slow transistor for power gating but may choose higher leakage transistors for logic since these will either be faster or can be run off a lower Vdd, hence reducing dynamic power.

The simple R plot illustrates the shift in operating frequency sweet spot (minimal total power) with higher leakage transistors. We considered leakage of 0.05 and 0.3 (arbitrary units). With low leakage it is best to compute slowly and finish just in time. With high leakage it is best to compute more quickly and then turn off for longer. A more-detailed analysis would reflect that designers today may have a choice of doping levels and hence transistor leakage at a given voltage and the performance also depends on the doping...

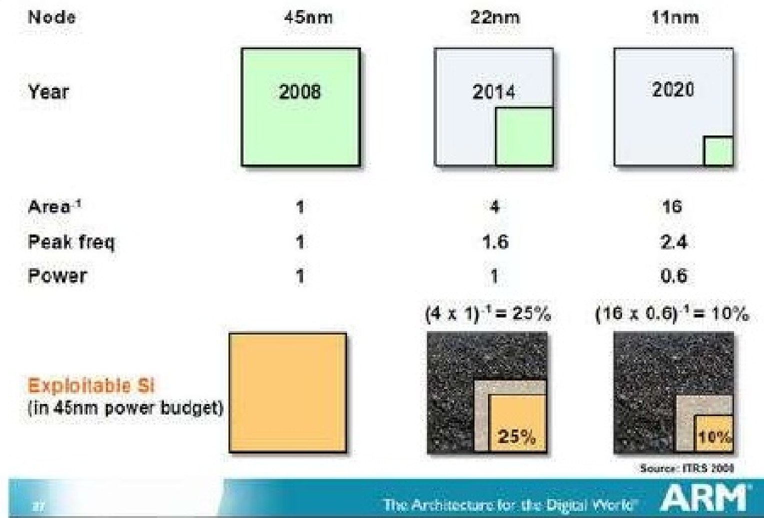
1.4.5 Future Trends

Transistors are still being made smaller and smaller.

We have hit the Power Wall resulting in a **Dark Silicon** approach. We can no longer turn on all of the chip and get the heat out cheaply: perhaps one tenth maximum for today's 22 nanometer chips. Even less in the future. Water cooling remains an option to mitigate the Power Wall.

Insights Article

- Systems increasingly limited by power consumption, not number of transistors
- ➔ "**Dark Silicon**" : Most of the chip will be OFF to meet thermal limits



Slow, bulky power transistors will turn thousands of power islands on and off under automated or manual control.

Conservation cores: use of high-level synthesis (HLS) of standard software kernels into application-specific hardware coprocessors and putting them on the chip in case they are needed? Afterall, they have negligable cost if not turned on. Venkatesh

KG 2 — Masked versus Reconfigurable Technology & Computing

There are different types of silicon chip and different types of manufacturer. Manufacturers can be broadly classified as:

1. So-called 'IDM' or 'vertical market' chip makers such as IBM and Intel that design, manufacture and sell their chips (Integrated Device Manufacturers).
2. Fabless manufacturers such as NVIDIA and Xilinx that design and sell chips but outsource manufacturing to foundry companies.
3. The foundry companies (such as TSMC and UMC) that manufacture chips designed and sold by their customers.

The world's major foundries are SMC and TSMC: Taiwan Semiconductor Manufacturing Company Limitedbut some verticals also provide fab services, perhaps even to competitors in the application space.

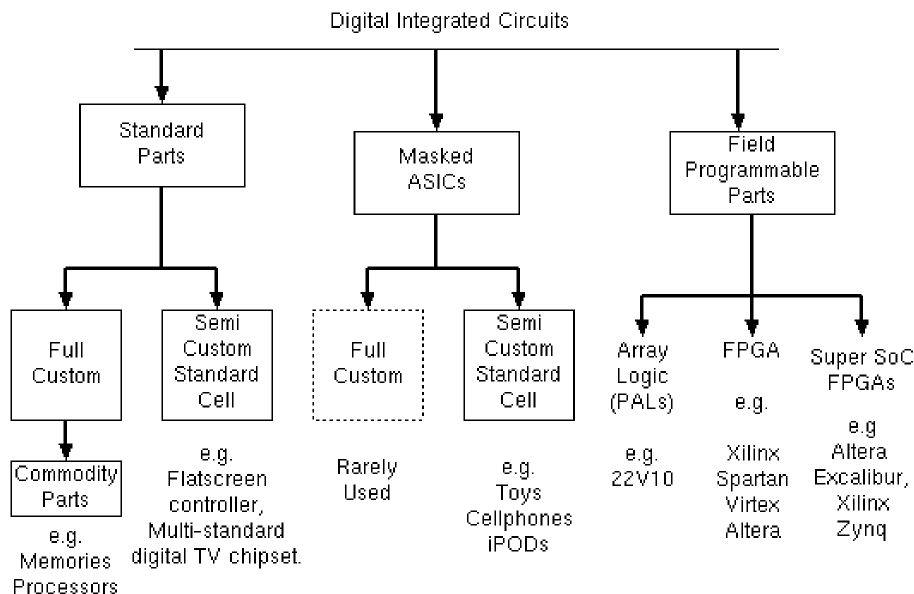


Figure 2.1: A rough taxonomy of digital integrated circuits.

Example Standard Cell Project: 8 Bit Adder0.5 Micron Cell Library

Figure 2.1 presents a historical taxonomy of chip design approaches. The top-level division is between standard parts, ASICs and field-programmable parts. Where a standard part is not suitable, the choice between full-custom and semi-custom and field-programmable approaches has to be made, depending on performance, production volume and cost requirements.

Additional notes:

There are deviations from this taxonomy: Complex PLDs cross between PALs and FPGA with low pin-to-pin delay. Structured ASICs were mask-programmed FPGAs popular around 2005. Today (2012-16), super FPGAs such as Zync are obliterating semi-custom masked ASICs for all but very-high-volume products. When Will FPGAs Kill ASICs?

Chips can be classified by function: Analog, Power, RF, Processors, Memories, Commodity: logic, discretes, FPGA and CPLD, SoC/ASIC, Other high volume (disk drive, LCD, ...).

2.0.6 Chip Types and Classifications

An SSD drive, such as the one in figure 2.2, originally used only standard parts: a microprocessor/microcontroller with off-chip DRAM and twenty FLASH chips. But the huge volume of production means that a custom microcontroller was soon preferred, but the FLASH chips themselves remain standard parts.

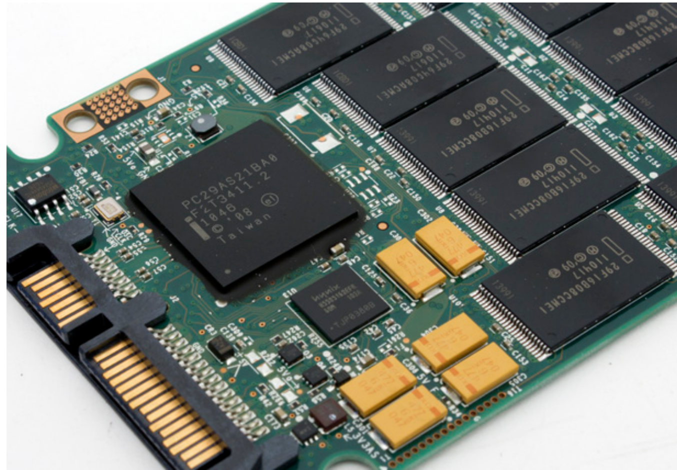


Figure 2.2: FLASH is now replacing spinning media in all but archival applications.

2.0.7 Standard Parts

A **standard part** is essentially any chip that a chip manufacturer is prepared to sell to someone else along with a datasheet and EDA (electronic design automation) models. The design may actually previously have been an ASIC for a specific customer that is now on general release. Many standard parts are general-purpose logic, memory and microprocessor devices. These are frequently full-custom designs designed in-house by the chip manufacturer to make the most of in-house fabrication line, perhaps using optimisations not made available to others who use the line as a foundry. Other standard parts include graphics controllers, digital TV chipsets, GPS receivers and miscellaneous useful chips needed in high volume.

2.0.8 Masked ASICs.

A masked ASIC (application-specific integrated circuit) is a device manufactured for a customer involving a set of masks where at least some of the masks are used only for that device. These devices include full-custom and semi-custom ASICs and masked ROMs.

A full-custom chip (or part of a chip) has had detailed, manual design effort expended on its circuits and the position of each transistor and section of interconnect. This allows an optimum of speed and density and power consumption.

Full-custom design is used for devices which will be produced in very large quantities: e.g. millions of parts where the design cost is justified. Full-custom design is also used when required for performance reasons. Microprocessors, memories and digital signal processing devices are primary users of full-custom design.

In semi-custom design, each cell has a fixed design and is repeated each time it is used, both within a chip and across many devices which have used the library. This simplifies design, but drive power of the cell is not optimised for each instance.

Semi-custom is achieved using a library of logic cells and is used for general-purpose VLSI design.

2.0.9 ASIC - Application-Specific Integrated Circuit

The cost of developing an ASIC has to be compared with the cost of using an existing part or an FPGA. The existing part may not perform the required function exactly, requiring either a design specification change, or some additional *glue logic* to adapt the part to the application.

More than one ASIC may be needed under any of the following conditions:

- application-specific functions are physically distant,
- application-specific functions require different technologies,
- application-specific functions are just too big for one ASIC,
- it is desired to split the cost and risk or reuse part of the system later on.

Factors to consider on a per-chip basis:

- power consumption limitation (powers above 5 Watts need special attention),
- die size limitation (above 11 mm on a side might escalate cost per mm^2),
- speed of operation — clock frequencies above 1 GHz raise issues,
- special considerations :
  - special static or dynamic RAM needs
  - analogue parts - what is compromised if these are integrated onto the ASIC ?
  - high power/voltage output capabilities for load control: e.g. motors.
- availability of a developed module for future reuse.

2.0.10 Semi-custom (cell-based) Design Approach

Standard cell designs use a set of well-proven logic cells on the chip, much in the way that previous generations of standard logic have been used as board-level products, such as Texas Instruments' System 74.

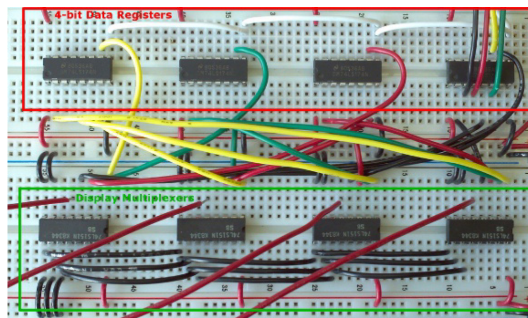
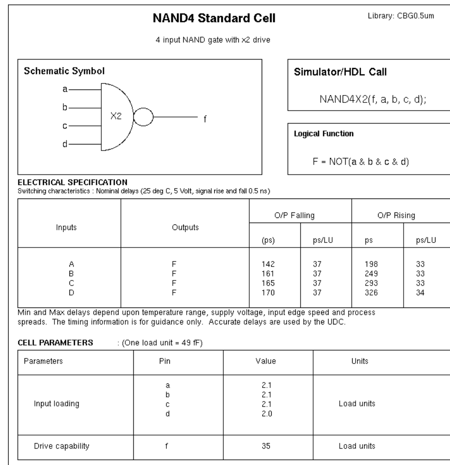


Figure 2.3: Discrete Logic Gates: Semi-custom design puts millions of them all on one die.

A library of standard logic functions is provided. Cells are placed on the chip and wired up by the user, in the same way that chips are placed on the PCB.

- Standard Cell - free placement and free routing of nets,
- Gate Array - fixed placement, masked or electrical programmable wiring.



NAND4 Standard Cell

Library: CBG0.5um

4 input NAND gate with x2 drive

Schematic Symbol

Simulator/HDL Call

NAND4X2(f, a, b, c, d);

Logical Function

F = NOT(a & b & c & d)

ELECTRICAL SPECIFICATION

Switching characteristics : Nominal delays (25 deg C, 5 Volt, signal rise and fall 0.5 ns)

Inputs	Outputs	O/P Falling		O/P Rising	
		(ps)	ps/LU	ps	ps/LU
A	F	142	37	198	33
B	F	161	37	249	33
C	F	165	37	293	33
D	F	170	37	326	34

Min and Max delays depend upon temperature range, supply voltage, input edge speed and process spreads. The timing information is for guidance only. Accurate delays are used by the UDC.

CELL PARAMETERS : (One load unit = 49 fF)

Parameters	Pin	Value	Units
Input loading	a	2.1	Load units
	b	2.1	
	c	2.1	
	d	2.0	
Drive capability	f	35	Load units

Figure 2.4: Typical cell data sheet from a standard cell library.

Figure 2.4 shows a cell from the data book for a standard cell library. This device has twice the 'normal' drive power, which indicates one of the compromises implicit in standard cell over full-custom, which is that the size (driving power) of transistors used in a cell is not tuned on a per-instance basis.

Mask-programmed gate array has been mostly replaced with the field-programmed FPGA except for analog/mixed-signal niches, such the example from TRIAD

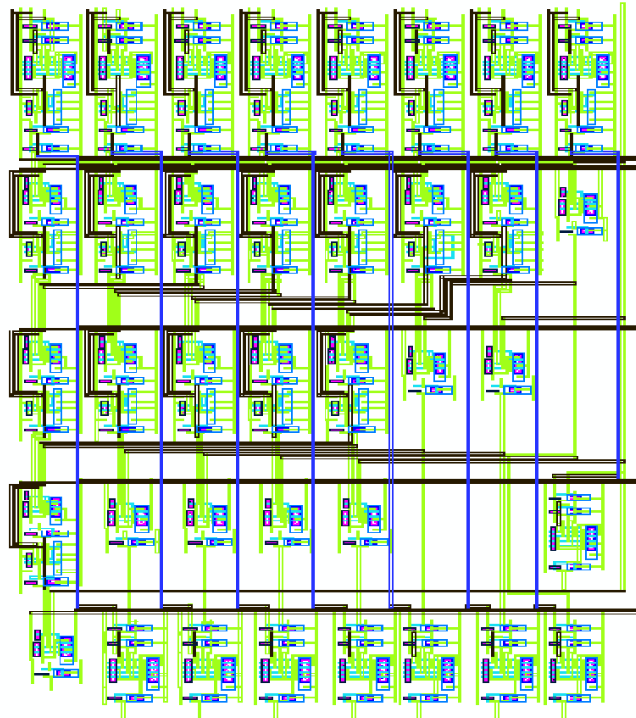


Figure 2.5: Standard cell layout for a Kogge-Stone adder. Taken from a student project (PDF on course web site).

In standard cell designs, cells from the library can freely be placed anywhere on the silicon and the number of IO pads and the size of the die can be freely chosen. Clearly this requires that all of the masks

used for a chip are unique to that design and cannot be used again. Mask making is one of the largest costs in chip design. (When) Will FPGAs Kill ASICs?

2.0.11 Cell Library Tour

In the lecture we will have a look at (some of) the following documents:

Standard Cell Student Project: Kogge Stone Adder

Cell libraries in the public domain: 0.5 Micron Cell LibraryAnother 90nm Cell LibrarySome others: VLSI TECHThings to note: there's a good variety of basic gates, including quite a few multi-level gates, such as AND-OR gate. There's also I/O pads, flip-flops and special function cells. Many gates are available with various output powers. For each gate there are comprehensive figures that enable one to predict its delay and energy use, taking into account its track loading, how many other gates it is feeding and the current supply voltage.

2.0.12 ASIC Costs: RE and NRE.

The cost of a chip splits into two parts: non-recurring engineering (NRE) and per-device cost.

Item	Cost (KUSD)	Total (KUSD)
NRE: 6 months : 10 H/W Engineers	250 pa	1250
NRE: 12 months : 20 S/W Engineers	200 pa	4000
NRE: 1 Mask set (45nm)	3000	3000
RE:An 8 inch wafer	5	5n
TOTAL	5	8125 + 5n

For small quantities: share cost of masks with other designs e.g. the MOSIS programme offers multiproject wafer (MPW).

2.0.13 Chip cost versus area

The per-device cost is influenced by the yield — the fraction of working dice. The fraction of wafers where at least some of the die work is the '**wafer yield**'. Historically yields have been low, but was typically close to 100 percent for mature 90 nm fabrication processes, but has again be a problem with smaller geometries in recent years.

The fraction of die which work on a wafer (often simply the '**yield**') depends on wafer impurity density and die size. Die yield goes down with chip area. The fraction of devices which pass wafer probe (i.e. before the wafer is diced) and fail post packaging tests is very low. However, full testing of analog sections or other lengthy operations are typically skipped at the wafer probe stage.

Assume processed wafer sale price might be 5000 dollars: A six inch diameter wafer has area ($3.14r^2$) = 18000 mm^2. A chip has area A, which can be anything between 2 to 200 mm^2 (including scoring lines). Dies per wafer is $18000/A$.

Probability of working = wafer yield × die yield (assume wafer yield is 1.0 or else included in the wafer cost).

Assume 99.5 percent of square millimetres are defect free. Die yield is then

$$P(\text{All A squares work}) = 0.995^A$$

cost of working dice is

$$\frac{5000}{\frac{18000}{A}0.995^A} \text{ dollars each.}$$

Cost of a working die given a six inch wafer with a processing cost of 5000 dollars and a probability of a square millimetre being defect free of 99.55 percent.

Area	Wafer dies	Working dies	Cost per working die
2	9000	8910	0.56
3	6000	5910	0.85
4	4500	4411	1.13
6	3000	2911	1.72
9	2000	1912	2.62
13	1385	1297	3.85
19	947	861	5.81
28	643	559	8.95
42	429	347	14.40
63	286	208	24.00
94	191	120	41.83
141	128	63	79.41
211	85	30	168.78
316	57	12	427.85
474	38	4	1416.89

For a chip with regular structure, such as a memory or an FPGA, additional hidden capacity can be deployed by burning fusible straps (aka links) during wafer probe test. This increases yield despite the larger area shipped in defect-free dies. AMD marketed a range of 3-core CPUs where the 4th, present on the die, had been strapped off.

2.0.14 Gate Arrays and Field-Programmable Logic.

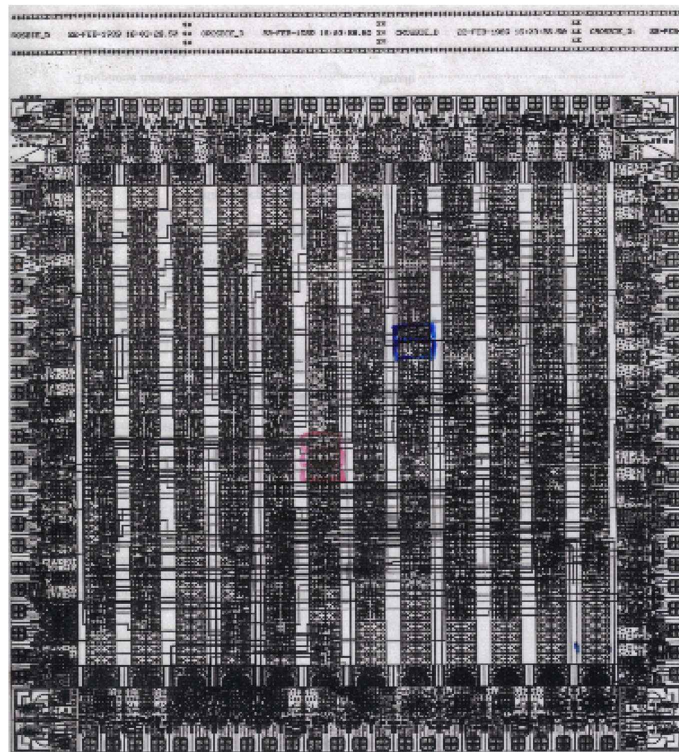


Figure 2.6: Mask for a Mask-Programmed Gate Array: (Greaves 1995, ECL for Ring Network)

Figure 2.6 reveals the regular layout of a masked gate array showing bond pads around the edge and wasted silicon area (white patches). A gate array comes in standard die sizes containing a fixed layout of configurable cells. Historically, there were two main forms of gate array:

- Mask Programmable,
- Field Programmable (FPGA).

In gate array designs, the silicon vendor offers a range of chip sizes. Each size of chip has a fixed layout and the location of each transistor, resistor and IO pad is common to every design that uses that size. Gate arrays are configured for a particular design by wiring up the transistors, gates and other components in the desired way. Many cells will be unused. For mask-programmed devices, the wiring up was done with the top two or three layers of metal wiring. Therefore only two or three custom masks were needed be made to make a new design. In FPGAs the programming is purely electronic (RAM cells control pass transistors).

The disadvantage of gate arrays is their intrinsic low density of active silicon. This arises from rounding up to the next available die size and the area overhead to support programming. The programming area overhead is especially severe for the FPGA.

2.0.15 FPGA - Field Programmable Gate Array

About 25 to 40 percent of chip sale revenue now comes from field-programmable logic devices. These are chips that can be programmed electronically on the user's site to provide the desired function. PALs and CPLDs are forms of programmable logic that are fast and small. But the most important form today is the FPGA.

Recall the Xilinx/Altera FPGA parts used in the Part IB E+A classes. Field-programmable devices may be volatile (need programming every time after power up), reprogrammable or one-time programmable. This depends on how the programming information is stored inside the devices, which can be in RAM cells or in any of the ways used for ROM, such as electrostatic charge storage (e.g. FLASH).

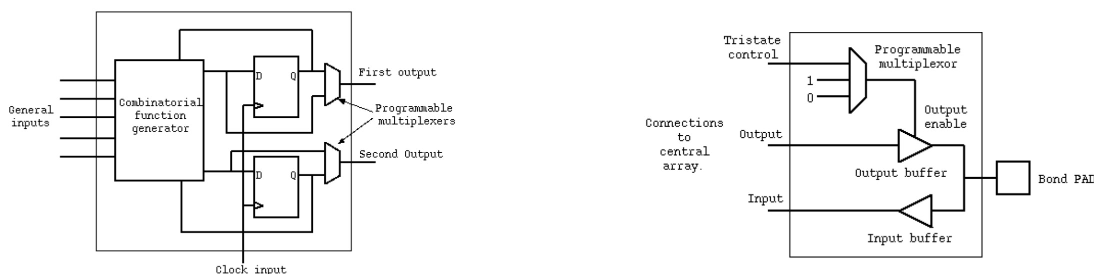
Except for niche applications (such as GaAs instead of Si), FPGAs are now always used instead of masked gate arrays and are starting to kill ASCIs (see link above).

Example: The part Ib practical classes use FPGAs from Altera: ECAD and Architecture Practical Classes

Summary data for a Virtex 5 Xilinx FPGA:

Part number	XC5VLX110T-2FFG1136C
Vendor	Xilinx Inc
Category	Integrated Circuits (ICs)
Number of Gates	110000
Number of I /O	640
Number of Logic Blocks/Elements	8640
Package / Case	1136-FCBGA
Operating Temperature	0C 85C
Voltage - Supply	1.14 V 3.45 V

Circa 2009, 65 nm technology, 6-input LUT, 64 bit D/P RAMs. Today Xilinx has the Virtex 7 series that includes the Zync SoC (of which more later) wikipedia: Virtex FPGA



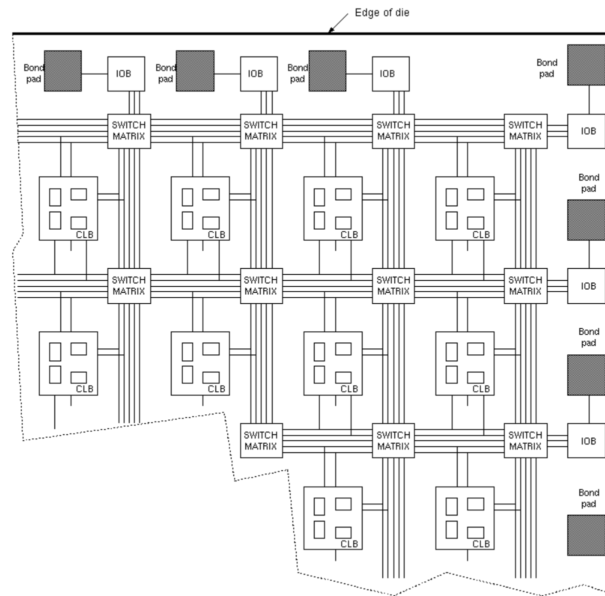


Figure 2.7: Field-programmable gate array structure, showing I/O blocks around the edge, interconnection matrix blocks and configurable logic blocks. In recent parts, the regular structure is broken up by custom blocks, including RAMs and multiplier (aka DSP) blocks.

An FPGA (field-programmable gate array) consists of an array of configurable logic blocks (CLBs), as shown in Figure 2.7. Not shown is that the device also contains a good deal of hidden logic used just for programming it. Some pins are also dedicated to programming. Such FPGA devices have been popular since about 1990.

Each CLB (configurable logic block) or slice typically contains two or four flip-flops, and has a few (five shown) general purpose inputs, some special purpose inputs (only a clock is shown) and two outputs. The illustrated CLB is of the look-up table type, where the logic inputs index a small section of pre-configured RAM memory that implements the desired logic function. For five inputs and one output, a 32 by 1 SRAM is needed. Some FPGA families now give the designer write access to this SRAM, thereby greatly increasing the amount of storage available to the designer. However, it is still an expensive way to buy memory.

FPGAs also soon started to contain RAM blocks (called block RAM or BRAM) and multiplier blocks called DSP (digital signal processing) blocks. The BRAM and DSP blocks are automatically deployed by the design tools by matching specific patterns in the user's RTL when coded appropriately. Today's FPGAs also contain many other 'hard-IP' blocks, such as PCIe, Ethernet and USB controllers that need to be manually instantiated as structural components in the RTL.

FPGAs tend to be slow, achieving perhaps one third of the clock frequency of a masked ASIC, owing to larger die area and because the signals pass through the programmable wiring junctions.

The Xilinx DSP block mostly contains a multiplier that delivers a 48 bit result and an adder for accumulating results where the output from one block has a high-performance programmable connection to a neighbour. The multiplier operands are two's complement, 25 and 18 bit operands. *Exercise: How many DSP blocks are needed for a 32x32 multiplier? What is it's latency? What differences does it make if only 32 bits of the result are needed?*

2.0.16 Circuit Switching

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

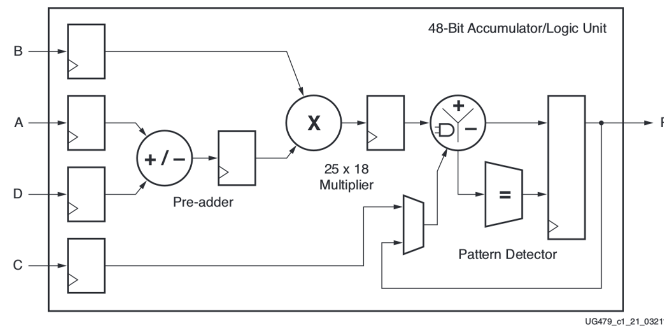


Figure 2.8: So-called DSP block in Xilinx Virtex 7 ((C) Xilinx Inc).

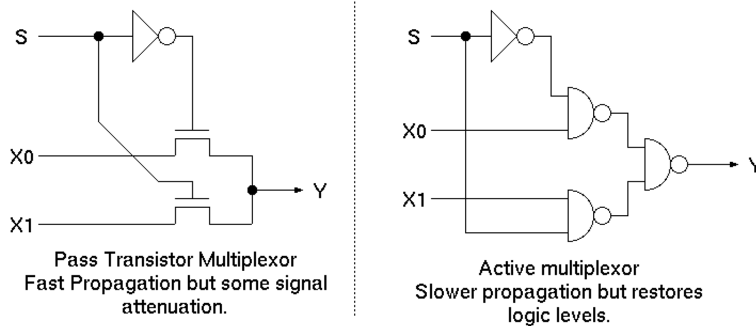


Figure 2.9: Pass transistor multiplexor compared with active path multiplexor.

FPGAs dominate the recent history of reconfigurable computing but are fine-grain owing to heritage in hardware circuits. There is an argument for having wider busses as the lowest programmable feature, which amortises the programming overhead to some extent, and yields the CGRA - coarse grain reconfigurable array.

2.0.17 Architectural Design: Partition and Exploration

A collection of algorithms and functional requirements must be implemented using one or more pieces of silicon. Each major piece of silicon contains one or more custom or standard microprocessors. Some silicon is custom for a high-volume product, some is shared over several product lines and some is third party or standard parts. The partition decisions take into account various aspects: fundamental silicon capabilities, stability of requirements, market forces, ease of reuse ...

Design Partition: Deciding on the number of processors, number of custom processors, and number of custom hardware blocks. The **system architect** must make make these decisions. SystemC helps them rapidly explore various possibilities.

Co-design and co-synthesis: two basic methods (can do different parts of the chip differently):

- **Co-design**: Manual partition between custom hardware and software for various processors,
- **Co-synthesis**: Automatic partitioning: simple 'device drivers' and inter-core message formats are created automatically:

Co-synthesis is still not in mainstream use (2018). Example algorithm: MPEG compression:

- A-to-D capture to framestore,

- Colour space conversion (RGB->YUV),
- DCT transform and variable Q quantisation,
- Motion detection,
- Huffman encoding.

Can any of this be best done on a general purpose (say ARM) core ?

MPEG Encoding 1MPEG algorithm 2

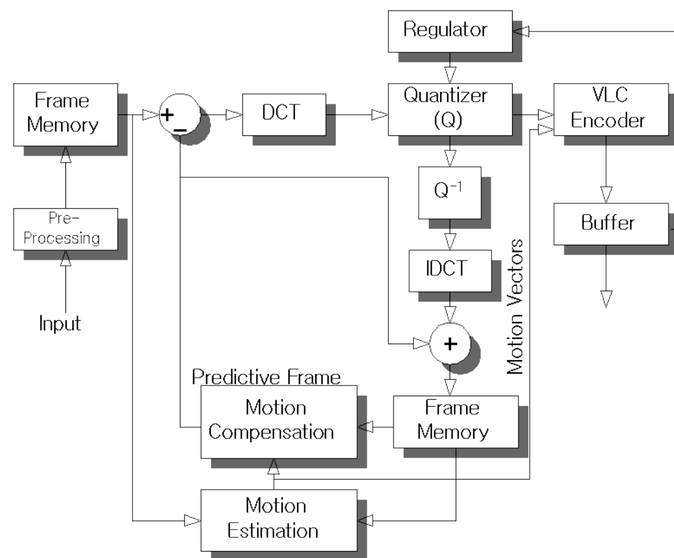


Figure 2.10: Flow chart for MPEG video compression.

2.0.18 H/W Design Partition

A number of separate pieces of silicon are combined to form the product. Reasons for H/W design partition:

- Modular Engineering At Large (Revision Control/Lifetime/Sourcing/Reuse),
- Size and Capacity (chips 6-11 mm in size),
- Technology mismatch (Si/GaAs/HV/Analog/Digital/RAM/DRAM/Flash)
- Supply chain: In-house versus Standard Part.
- Isolation of sensitive RF signals,
- Cost: a new chip spin of old IP is still very expensive.

2.1 H/W versus S/W Design Partition Principles

Many functions can be realised in software or hardware. Decide what to do in hardware:

- physical I/O (line drivers/transducers/media interfaces),

- highly compute-intensive, **fixed** functions,

what to do on custom processors or with custom instructions/coprocessors on an extensible processor:

- bit-oriented operations,
- highly compute-intensive SIMD,
- other algorithms with custom data paths,
- algorithms that might be altered post tape out.

and what to do in S/W on standard cores:

- highly-complex, non-repetitive functions,
- low-throughput computations of any sort,
- functions that might be altered post tape out,
- generally, as much as possible.

Custom processor synthesis commercial offering: See (link updated) Tensilica Customizable Processor and DSP IP

When designing a sub-system we must choose what to have as hardware, what to have as software and whether custom or standard processors are needed. When designing the complete SoC we must think about sharing of sub-system load over processors. Example: if we are designing a digital camera, how many processors should it have and can the steadicam and motion estimation processing be done in software ? Would a hardware implementation use less silicon and less battery power?

- The functions of a system can be expressed in a programming language or similar form and this can be compiled fully to hardware or left partly as software
- Choosing what to do in hardware and what to do in software is a key decision. Hardware gives speed (throughput) but software supports complexity and flexibility.
- Partitioning of logic over chips or processors is motivated by interconnection bandwidth, raw processing speed, technology and module reuse.

2.1.1 Typical Radio/ Wireless Link Structure.

Radio communication above the VHF frequency range (above 150 MHz) uses high-frequency waveforms that cannot be directly processed by A-to-D or D-to-A technology. Hetrodyning is analogue multiplication with a sine wave carrier to perform frequency conversion. This exploits the sin(A)*sin(B) = -cos(A+B)/2 part of the standard trig identity for converting upwards and the other half for converting downwars.

The high frequency circuity is almost always implemented on a separate chip from the digital signal processing (DSP) for the baseband logic. The radio transmitter is typically 50 percent efficient and will use a about 100 mW for most indoor purposes. A cell phone transmitter has a maximum power of 4W which will be used when a long distance from the mast. (Discuss: Having a mast in a school playground means the children are beaming far less radio signal from their own phones into their own heads.) The backlight on a mobile phone LCD may use 300mW (100 LEDs at 30 mW each).

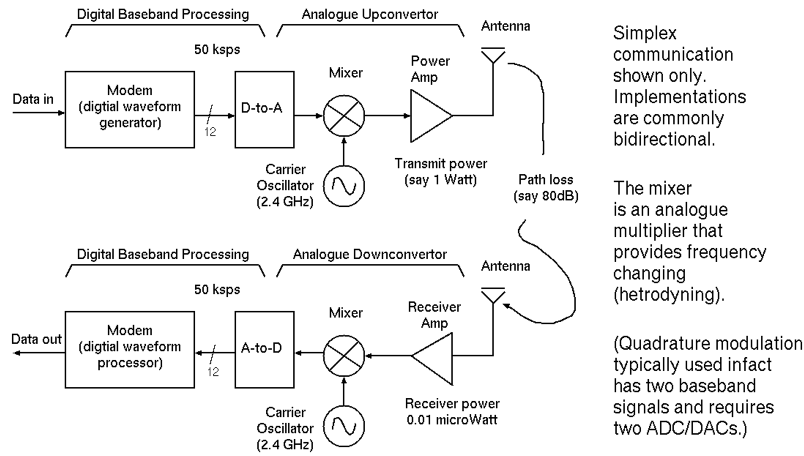


Figure 2.11: Typical structure of modern simplex radio link.

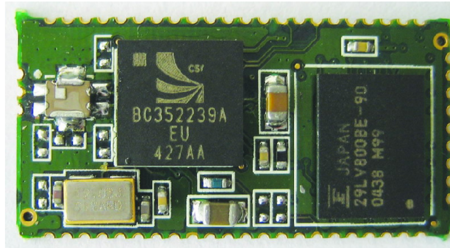


Figure 2.12: Broadcom (Cambridge Silicon Radio) Bluetooth Module circa 2000.

2.1.2 Partitioning example: A Bluetooth Module.

An initial implementation of the Bluetooth radio was made of three pieces of silicon bonded onto a small fibreglass substrate...

An initial implementation of the Bluetooth radio was made of three pieces of silicon bonded onto a small fibreglass substrate with overall area of 4 square centimetres. The module was partitioned into three pieces of silicon partly because the overall area required would give a low yield, but mainly because the three sections used widely different types of circuit structure.

The analog integrated circuit contained amplifiers, oscillators, filters and mixers that operate in the 2.4 GHz band. This was too fast for CMOS transistors and so bipolar transistors with thin bases were used. The module amplifies the radio signals and converts them using the mixers down to an intermediate frequency of a few MHz that can be processed by the ADC and DAC components on the digital circuit.

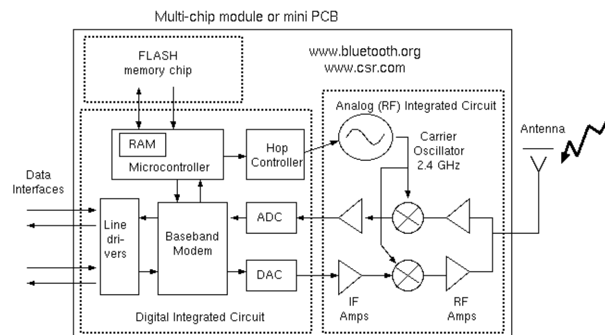


Figure 2.13: Example of a design partition — Block diagram of Bluetooth radio module (circa 2000).

Figure 2.14: WiFi laptop module. Shielding lid, shown left, has been lifted off.

The digital circuit had a small amount of low-frequency analog circuitry in its ADC and DACs and perhaps in its line drivers if these are analog (e.g. HiFi). However, it was mostly digital, with random logic implementations of the modem functions and a microcontroller with local RAM. The local RAM holds a system stack, local variables and temporary buffers for data being sent or received.

The FLASH chip is a standard part, non-volatile memory array that can hold firmware for the microcontroller, parameters for the modem and encryption keys and other end application functions. The flash memory is a standard 29LV800BE (Fujitsu) - 8m (1m X 8/512 K X 16) Bit

Today, the complete Bluetooth module can be implemented on one piece of silicon, but this still presents a major technical challenge owing to the diverse requirements of each of the sub-components.

2.2 Super FPGAs: Example Xilinx Zynq

We use the terms hard and soft to differentiate between functions are are determined by the fabrication masks and are loaded into the programmable fabric. Although it was common to put so-called **soft** CPU cores in the programmable logic, today's devices have hardened CPUs and many other **hard** IP blocks. Connecting DRAM to FPGAs has become a common requirement in the last decade and hardened DRAM controllers are now common.

The high cost of ASIC masks now makes FPGA suitable for most medium volume production runs (e.g sub 10,000 units) which includes most recording studio equipment and passenger-in-the-road detection for high-end cars. The dark silicon trend means we can put all IP blocks on one chip provided we leave them mostly turned off.

The Zynq from Xilinx has two ARM cores, all the standard SoC IP blocks and an area of FPGA programmable logic, all on one die. The same DRAM bank is accessible to both the hardened ARMs and the programmable logic.

Xilinx Zynq-7000 Product Brief (PDF)

Flexible I/O routing means physical pads can be IP block bond outs, GPIOs or FPGA I/O blocks.

Vital statistics for the first Zynq offerings:

Figures for some more-recent devices:

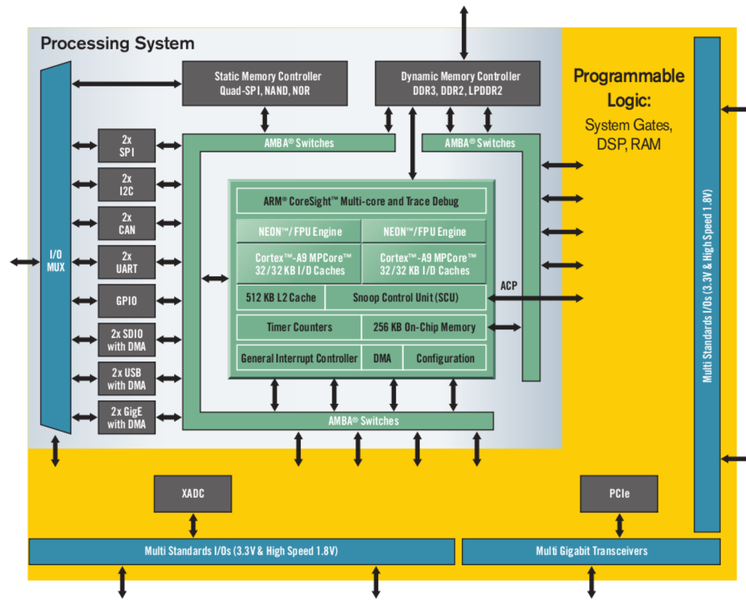


Figure 2.15: Xilinx Zynq 7000 Overview.

	Zynq-7000 Product Table (Hardware View)				
	Device Name	Z-7010	Z-7020	Z-7030	Z-7045
	Part Number	XC7Z010	XC7Z020	XC7Z030	XC7Z045
	Processing System (Dual ARM® Cortex™-A9 MPCore™ with NEON™ & Double Precision FPUCache, Memory Controllers, DMA, Security and Peripherals)	Same processing system for all devices			
Programmable Logic	Xilinx 7 Series Programmable Logic Equivalent	Artix™-7 FPGA	Artix™-7 FPGA	Kintex™-7 FPGA	Kintex™-7 FPGA
	Programmable Logic Cells (Approximate ASIC Gates(2))	28K Logic Cells (~430K)	85K Logic Cells (~1.3M)	125K Logic Cells (~1.9M)	350K Logic Cells (~5.2M)
	Logic Cells	28,160	85,120	125,760	349,760
	Look-Up Tables LUTs	17,600	53,200	78,600	218,600
	Flip Flops	35,200	106,400	157,200	437,200
	Extensible Block RAM (# 36 Kb Blocks)	240 KB (60)	560 KB (140)	1,060 KB (265)	2,180KB (545)
	Programmable DSP Slices (18x25 MACCs)	80	220	400	900
	Peak DSP Performance (Symmetric FIR)	58 GMACS	158 GMACS	480 GMACS	1080 GMACS
	PCI Express® (Root Complex or Endpoint)	—	—	Gen2 x4	Gen2 x8
	Agile Mixed Signal (AMS)/XADC	2x 12 bit, 1 MSPS ADCs with up to 17 Differential Inputs			
	Security (1)	AES and SHA 256b for secure configuration			

Figure 2.16: Xilinx Zynq 7000 FPGA Resources.

Device Name	VU31P	VU33P	VU35P	VU37P	VU11P	VU13P
System Logic Cells (K)	962	962	1,907	2,852	2,835	3,780
CLB Flip-Flops (K)	879	879	1,743	2,607	2,592	3,456
CLB LUTs (K)	440	440	872	1,304	1,296	1,728
Max. Distributed RAM (Mb)	12.5	12.5	24.6	36.7	36.2	48.3
Total Block RAM (Mb)	23.6	23.6	47.3	70.9	70.9	94.5
UltraRAM (Mb)	90.0	90.0	180.0	270.0	270.0	360.0
HBM DRAM (GB)	4	8	8	8	–	–
HBM AXI Interfaces	32	32	32	32	–	–
Clock Mgmt Tiles (CMTs)	4	4	8	12	12	16
DSP Slices	2,880	2,880	5,952	9,024	9,216	12,288
Peak INT8 DSP (TOP/s)	8.9	8.9	18.6	28.1	28.7	38.3
PCIe® Gen3 x16 / Gen4 x8	4	4	5	6	3	4
CCIX Ports(1)	4	4	4	4	–	–
150G Interlaken	0	0	2	4	6	8
100G Ethernet w/ RS-FEC	2	2	5	8	9	12
Max. Single-Ended HP I/Os	208	208	416	624	624	832
GTY 32.75Gb/s Transceivers	32	32	64	96	96	128

Figure 2.17: Parts available from Xilinx in 2018

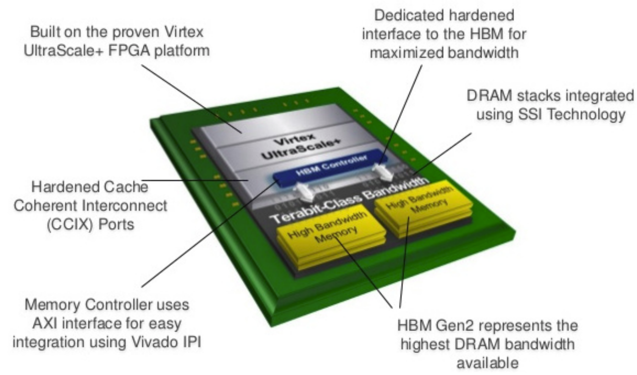


Figure 2.18: FPGA connected closely to several DRAM chips in Multi-chip Module (MCM).

KG 3 — Custom Accelerator Structures

Perhaps the first hardware accelerators added to alongside the integer execution units of early computers were for floating-point arithmetic. But accelerators can serve many different purposes and sit elsewhere within the architecture. In this section we note the design considerations.

3.1 H/W to S/W Interfacing Techniques

The term 'Programmed I/O' can refer to either MMIO or PMIO. These are the main alternative to I/O performed by DMA. A note on terms:

- Port-mapped I/O (PMIO) refers to a special address space outside of normal memory that is accessed with instructions such as IN and OUT.
- Memory-mapped I/O (MMIO) refers to I/O devices being allocated addresses inside the normal Von Neumann address space that is primarily used for program and data. Such I/O is done using instructions such as LOAD and STORE.

PMIO was very useful on A16 microprocessors since valuable address space was not consumer by the I/O devices, but A32 architectures generally provide no PMIO instructions and hence use MMIO.

An accelerated system is divided into some number of hardware and software blocks with appropriate means of interconnection. The primary ways of connecting hardware to software are:

- CPU coprocessor and/or custom instructions,
- Packet channel connected as coprocessor or mapped to main register file,
- Programmed I/O to pin-level GPIO register,
- Programmed I/O to FIFOs,
- Interrupts (hardwired to one core or dynamically dispatched),
- Pseudo-DMA: processor generates memory addresses or network traffic and the accelerator simply snoops or interposes on the data stream,
- DMA - Autonomous unit, much like a CPU in its own right.

Another design point is to do everything in hardware with no CPUs, but a CPU in a supervisorary role is normally sensible.

3.2 Custom Accelerators on SoC or ASIC

Suppose something like the following fragment of code is a dominant consumer of power in a portable embedded mobile device:

```
for (int xx=0; xx<1024; xx++)
{
   unsigned int d = Data[xx];
   int count = 0;
   while (d > 0) { if (d&1) count ++;   d >>= 1; }
   if (!xx || count > maxcount) { maxcount = count; where = xx; }
}
```

This kernel tallies the set bit count in each word: such bit-level operations are inefficient using general-purpose CPU instruction sets. This is about the smallest possible example for which a hardware accelerator might be worthwhile. Do you think it could be worthwhile?

A dedicated **hardware accelerator** avoids instruction fetch overhead and is generally more power efficient. Analysis using Amdahl's law and high-level simulation (SystemC TLM) can establish whether a hardware implementation is worthwhile. There are several feasible partitions:

1. Extend the CPU with a **custom datapath** and custom ALU (Figure 3.1a) for the inner tally function controlled by a **custom instruction**.
2. Add a tightly-coupled **custom coprocessor** (Figure 3.1b) with fast data paths to load and store operands from/to the main CPU. The main CPU still generates the address values `xx` and fetches the data as usual.
3. Place the whole kernel in a **custom peripheral unit** (Figure 3.2) with operands being transferred in and out using programmed I/O or pseudo-DMA.
4. As 3, but with the new IP block having bus master capabilities so that it can fetch the data itself (DMA), with polled or interrupt-driven synchronisation with the main CPU.
5. Use an FPGA or bank of FPGAs without a conventional CPU at all ... (see later).

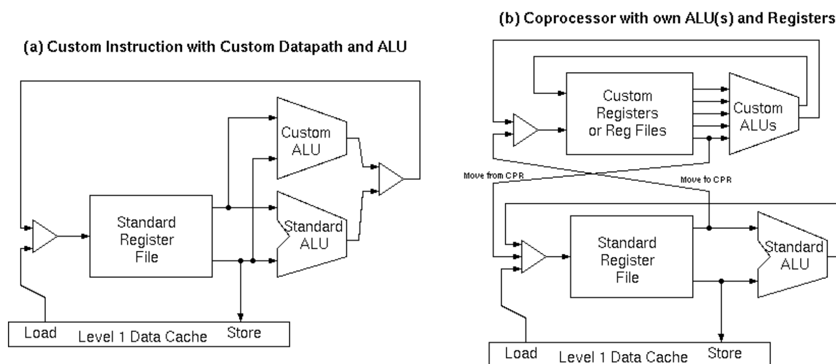


Figure 3.1: A custom ALU operation implemented in two similar ways: as a custom instruction or as a coprocessor.

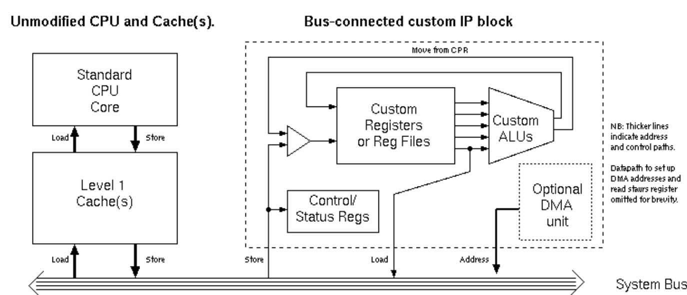


Figure 3.2: A custom function implemented as a peripheral IP block, with optional DMA (bus master) capability.

The special hardware in all approaches may be manually coded in RTL or compiled using HLS from the original C implementation.

In the first two approaches, both the tally and the conditional update of the maxcount variable might be implemented in the custom ALU, but most of the gain would come from the tally function itself and the detailed design might be different depending on whether custom instruction or coprocessor were used.

The custom instruction operates on data held in the normal CPU register file. The bit tally function alone reads one input word and yields one output word, so it easily fits within the addressing modes provided for normal ALU operations. Performing the update of both the maxcount and word registers in one custom instruction would require two register file writes and this may not be possible in one clock cycle and hence, if this part of the kernel is placed in the custom datapath we might lean more towards the co-processor approach.

Whether to use the separate, bus-connected IP block depends on whether the processor has something better to do in the meantime and that there is sufficient bus bandwidth for them both to operate. Whether the data is likely to be in or be needed in a given L1 data cache is also an important factor.

The separate IP block may or may not use a DMA controller. Given that the ARM now has a ones-tally instruction in its normal ALU, getting an ARM to move the data into the separate IP block may be a really poor design point.

With increasing available transistor count in the form of **dark silicon** (ie. switched off most of the time) in recent and future VLSI, implementing standard kernels as custom hardware cores is a possible future trend for power conservation. The **conservation cores** project Venkatesh considered implementing the inner loops of a 'mature computations' such as a selection of popular Android applications in silicon on future mobile phones.

3.3 Bump-in-Wire Reconfigurable Accelerator Architectures

FPGA is increasingly seen as a computing element alongside CPU and GPU. Energy savings of two orders of magnitude are often seen when a suitable application is accelerated on FPGA. Execution speed can also commonly increase, although this is hampered by the order-magnitude reduction in clock frequency compared with CPU (e.g 200 MHz instead of 2 GHz).

Historically, many hardware accelerator projects have ultimately been unsuccessful because: either

- The hardware development takes too long and general-purpose CPUs meanwhile progress and overtake (their development teams are vastly more resourced)
- The overhead of copying the data in and out of the accelerator exceeds the processing speed up.
- The hardware implementation is out of date, such as when the requirements or a protocol standard is changed.

But by implementing accelerators on FPGA at a place where the data is moving already, these problems can be largely mitigated. Also, until recently, FPGAs have not had hardened DRAM controllers and consequently been short of DRAM bandwidth.

Microsoft have had several generations of blade design for their data centres. Recent ones have placed the FPGA in series with blade's network connection, thereby enabling copy-free pre- and post-processing of data. For instance, an index hash can be computed on database fields.

The FPGAs on neighbouring cards are also locally interconnected with a high-speed ring or mesh network, enabling them to be pooled and managed independently of the blade's CPUs. This enables systolic sorting networks and the like to be formed; e.g. for keeping the k-best Bing search results.

The QPI interconnection between CPUs is cache-consistent. Some FPGA-accelerated blade designs connect the FPGA to such a cache-consistent interconnect.

On the Zynq platform (Figure 3.7) a number of methods for connecting to the reconfigurable logic are available - they are mostly via AXI ports. They vary in cache-consistency and bandwidth and initiator/target polarity. Of the initiating ports, both provide connection to the on-chip SRAM and the

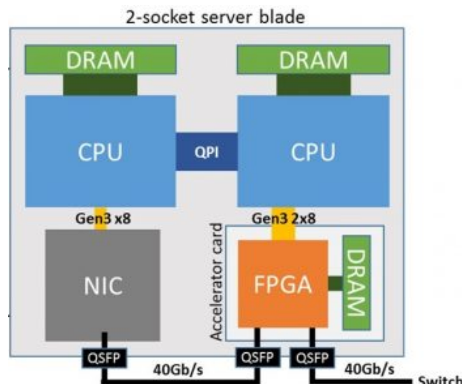


Figure 3.3: Bump-in-Wire design for Microsoft Catapult Accelerator (2016).

single DRAM bank that is also shared with the ARM cores, But one form is cache-coherent with the ARMs and the other is not, but has higher bandwidth.

3.3.1 FPGA Computing to replace Von Neumann Dominance?

The Von Neumann computer hit a wall in terms of increasing clock frequency. It is widely accepted that Parallel Computing is the most energy-efficient way forward. The FPGA is intrinsically massively-parallel and can exploit the abundant transistor count of contemporary VLSI. Andre DeHon points out that the Von Neumann architecture no longer addresses the correct problem: he writes 'Stored-program processors are about compactness, fitting the computation into the minimum area possible.'

Why is computing on an FPGA becoming a good idea ? 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. Standard computers support fixed data sizes only and only two encodings (integer and floating point), both of which can waste a lot of energy compared with better encodings [Constantinidies].
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 near-data-processing (but the FPGA overall size is x10 times larger (x100 area) owing to overhead of making it reconfigurable.

   So
9. The massively-parallel premise of the FPGA is the correct way forward, as indicated by asymptotic limit studies [DeHon].

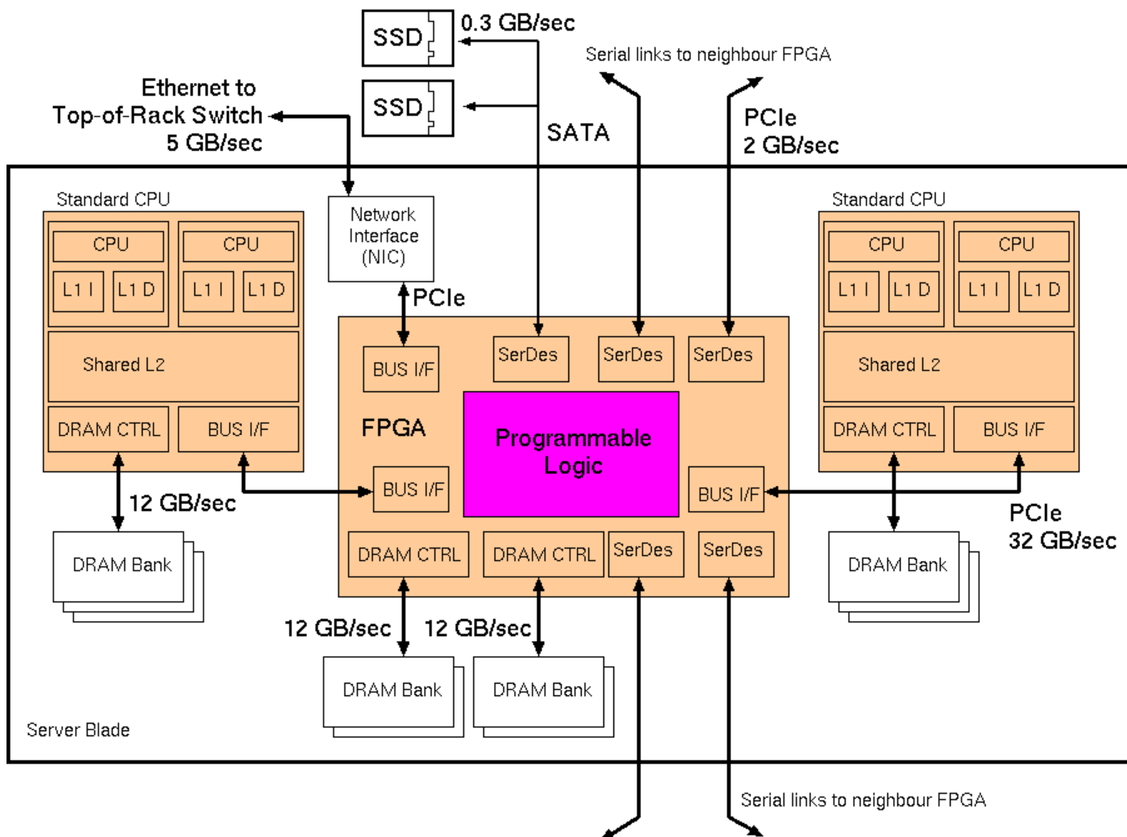


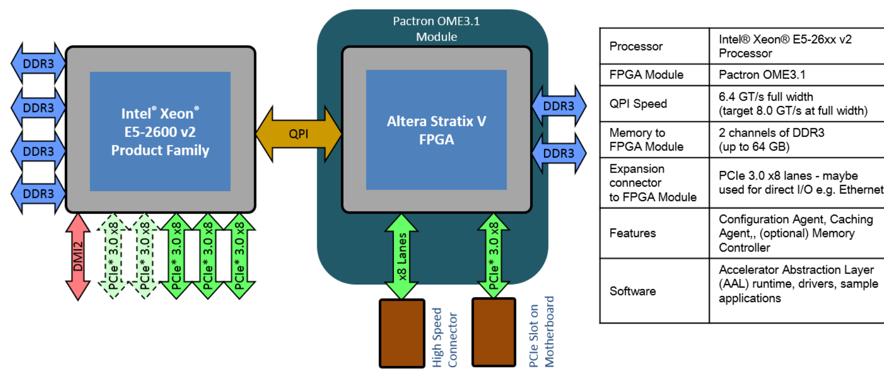
Figure 3.4: Representative 'bump-in-wire' server blade architecture that has the FPGA in series with Network Interface and Disk Drives.

Programming an FPGA has been a problem. As we shall discuss in a later section, end users cannot be expected to be hardware or RTL experts. Instead, new compiler techniques to port software-style programming to the FPGA are developing. The main approaches today are OpenCL and HLS.

3.3.2 FPGA As The Main Or Only Processor ?

The FPGA generally needs connecting to mainstream computing resources, if only for management or file system access. Figure 3.9 shows a typical setup using a Xilinx Zynq chip (from the Kiwi HLS project) KiwiThe so-called 'substrate' provides basic start/stop and debugging control, together with access to a programmed I/O (PIO) register file. It also provides services for a network-on-chip between the processing blocks that provides an 'FPGA Operating System' API for file server input and output. Finally, it also provides access to DRAM banks and platform-specific inter-FPGA links (not shown).

Figure 3.5: Catapult Blade - FPGA is at the lower right, its heatsink visible above its blue DRAM DIMMs.



Processor	Intel® Xeon® E5-26xx v2 Processor
FPGA Module	Pactron OME3.1
QPI Speed	6.4 GT/s full width (target 8.0 GT/s at full width)
Memory to FPGA Module	2 channels of DDR3 (up to 64 GB)
Expansion connector to FPGA Module	PCIe 3.0 x8 lanes - maybe used for direct I/O e.g. Ethernet
Features	Configuration Agent, Caching Agent,, (optional) Memory Controller
Software	Accelerator Abstraction Layer (AAL) runtime, drivers, sample applications

Figure 3.6: Cache-consistent interconnection between CPU and FPGA.

Will FPGA totally replace Von Neumann? Currently many problems remain: the static-schedule does not work well for all applications, the debugging support is in its infancy and compile times are often hours. The length overhead of programmable wiring adds a lot of capacitance, so only if the data ends up moving far less physical distance will energy be saved. Nonetheless, there are many success stories in areas such as automated trading, database indexing, deep packet inspection, genomics and cryptology. There are many further papers and PhDs to be written ...

3.3.3 Virtualising the FPGA

FPGA in the datacentre was first deployed by Microsoft for Bing acceleration. A user-programmable FPGA fabric as part of the Cloud offering was made available by Amazon at the start of 2017. Where customer designs are loadable into the FPGA, security issues arise. Overheating, denial of service and data theft must be avoided and a means to virtualise (share) the FPGA resources is needed.

The FPGA has much more user state to context switch than a conventional CPU, so a sub-Hertz context switch rate is likely. This is at least 3-orders coarser than typical CPU time sharing. All recent FPGA families support dynamic loading of selected parts of the die while the remainder continues operating. The reconfiguration time is typically 10 to 100 ms per zone. This perhaps opens the way to finer-grained sharing, if needed.

One approach to virtualisation is to use dynamically-loaded modules under the 'Shell' and 'Role' approach to FPGA real-estate (Figure 3.10) where the customer only has access to the part of the die via partial-reconfiguration. The service shim or shell provides a protected service layer. Disadvantages of this technique are that quite a lot of real estate (and especially the multiplier resource therein) is consumed with overhead. Also there are design fragmentation inefficiencies. And many of the FPGA resources, such as most of the I/O pins and hardened peripherals go unused (not a problem for a Dark Silicon future!).

3.3.4 Future FPGA Architectures for the Cloud

It is a possible that a future generation of FPGA devices will be developed specialised for server blades in the cloud. See Figure 3.11. The user logic will connect to the outside world only through a service network. Kiwi HLS uses such a NoC for O/S access, as does LEAP. But reconfigurable interconnection between adjacent zones that jointly host an application (blue) might be provided. Having the service network hardened makes it about 100-fold more area efficient, allowing much more complexity, security and monitoring. The local memory devices would also provide virtualisation assistance in their hardened controllers.

A simple NoC does not take up much FPGA resources, but the ideal NoC would probably support credit-based flow control and also include support for virtualisation.

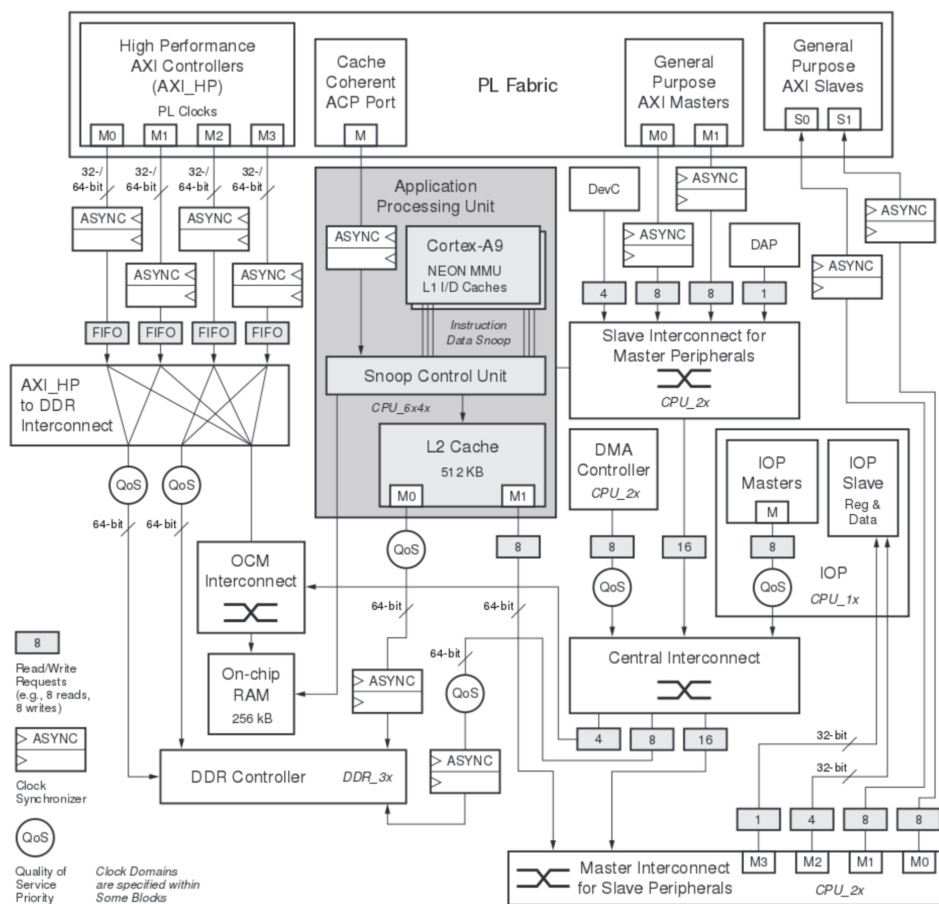


Figure 3.7: Block Diagram of the Xilinx Zynq Platform.

3.3.5 Links and Citations

Amazon, Microsoft, Alibaba and others are now offering FPGA in the cloud services. Some links:

Multi-FPGA System Integrator Intel/Altera Acceleration As A Service Reconfigurable Computing: Architectures and Design Methods. T.J. Todman 2005 Amazon EC2 F1 Instances

Evaluating the Energy Efficiency of Reconfigurable Computing Toward Heterogeneous Multi-Core Computing, Fabian Nowak

A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services, Andrew Putnam

Michael Dales Phd Thesis addressed this first: Managing a Reconfigurable Processor in a General Purpose Workstation Environment - Date 2003

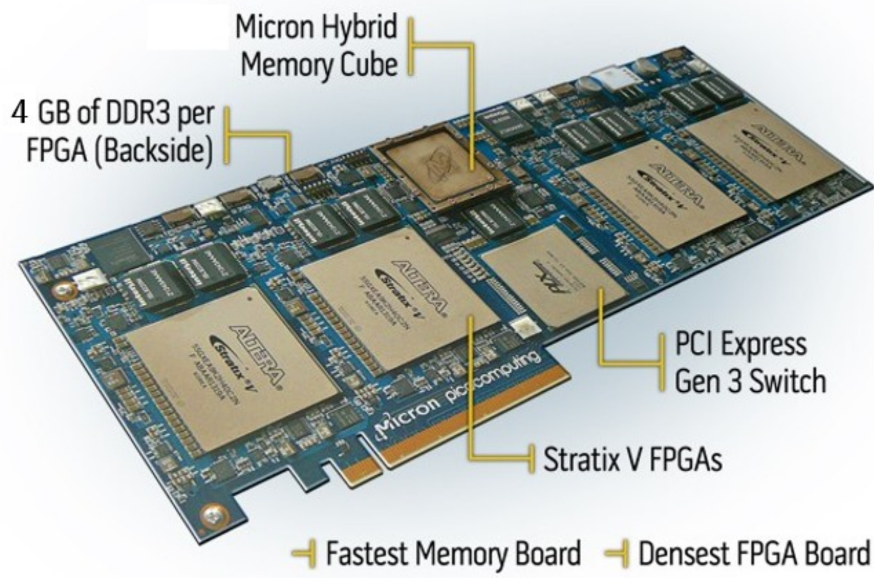


Figure 3.8: A CPU-less cloud sever blade from Pico Computing (4 FPGAs + Lots of DRAM).

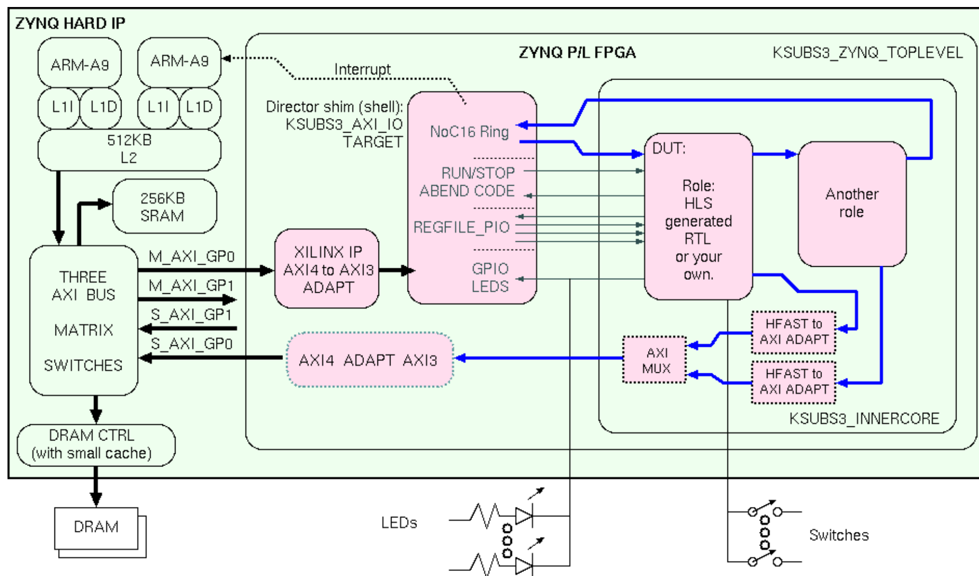


Figure 3.9: Block Diagram of the Kiwi HLS substrate, ksubs3, installed in a Zynq FPGA (all parts simplified).

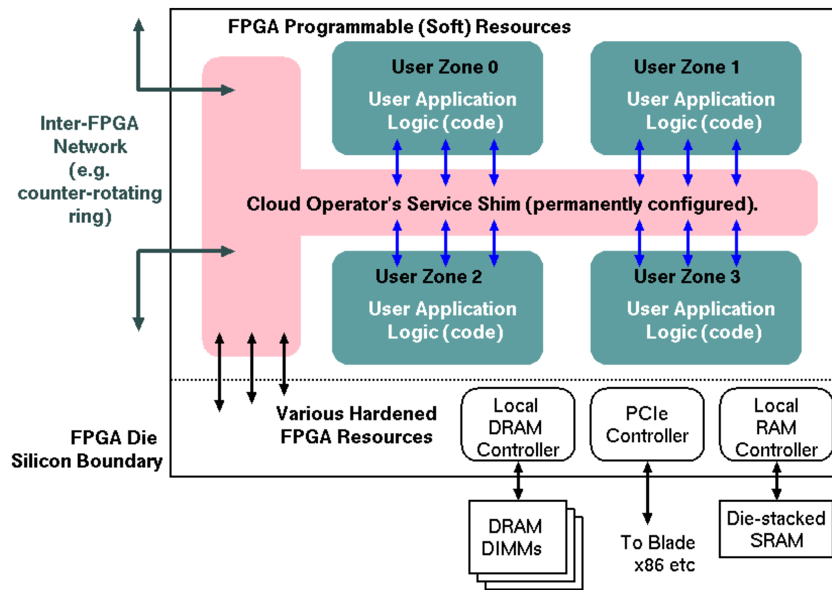


Figure 3.10: The reconfigurable logic resources of a conventional FPGA can be manually partitioned into user reconfigurable areas and a shell/shim for service access.

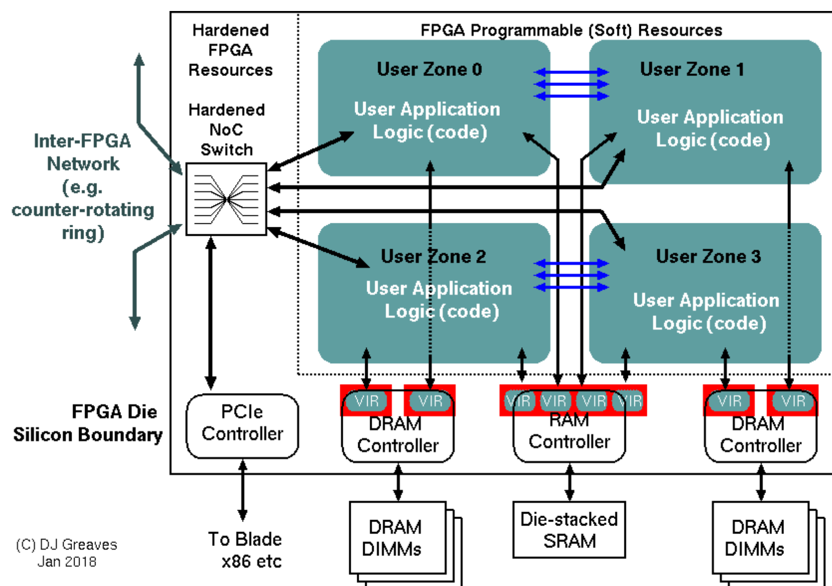


Figure 3.11: Putative block diagram for a cloud-oriented FPGA for scientific acceleration in the future.

KG 4 — RTL, Interfaces, Pipelining and Hazards

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

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

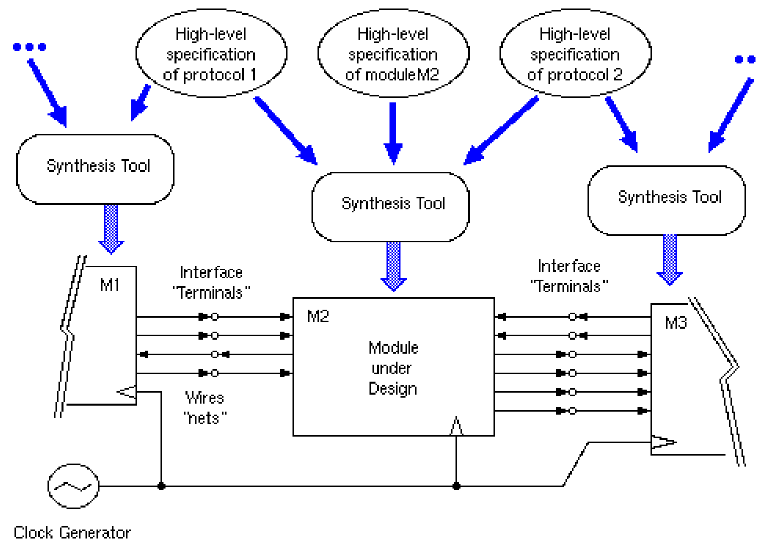


Figure 4.1: Generic (net-level) Module Interconnection Using Protocols and Interfaces.

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

4.1 Protocol and Interface

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

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

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

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

Additional notes:

There are four conceivable clock strategies for an interface:

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

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

4.1.1 Transactional Handshaking

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

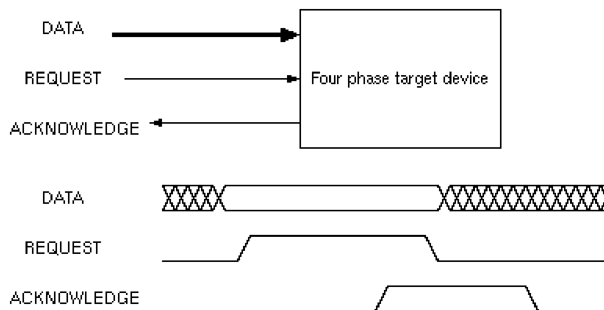


Figure 4.2: Timing diagram for an asynchronous, four-phase handshake.

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

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

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

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

4.1.2 Transactional Handshaking in RTL (Synchronous Example)

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

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

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

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

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

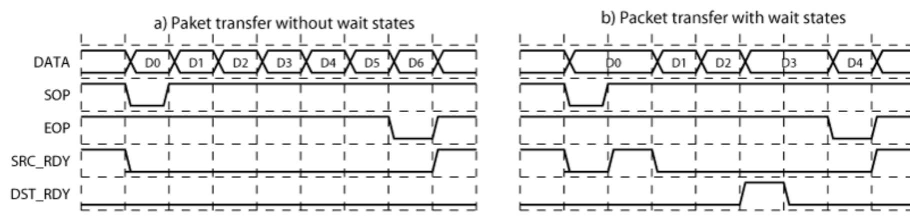


Figure 4.3: Timing diagram for the synchronous LocalLink protocol.

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

Additional notes:

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

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


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

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

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


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

And here is a corresponding data sink:

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

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

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

endmodule // LocalLinkSrc
```

Additional notes:

And here is a testbench that wires them together:

```
module SIMSYS();

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

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

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



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

endmodule // SIMSYS
```

4.2 Architecture: Bus and Device Structure

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

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

- Speed of light on silicon and on a PCB is 200 metres per microsecond.
- A clock frequency of 2 GHz has a wavelength of 2E8/2E9 = 10 cm.
- Within a synchronous digital clock domain we require connections to be less than (say) 1/10th of a wavelength.
- Conductor series resistance further slows signal propagation and is dominant source of delay.
- So need to register a signal in several D-types if it passes from one corner of an 8mm chip to the other!

- Can have several thousand wires per millimetre per layer: fat busses (128 bits or wider) are easily possible.
- Significant DRAM is several centimetres0 away from the SoC and also has significant internal delay.

Hence we need to use protocols that are tolerant to being registered (passed through D-type pipeline stages). The four-phase handshake has one datum in flight and degrades with reciprocal of delay. We need something a bit like TCP that keeps multiple datums in flight. (Die stacking and recent DRAM-on-SoC approaches reduce wire length to a few mm for up to 500 MB of DRAM.)

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

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

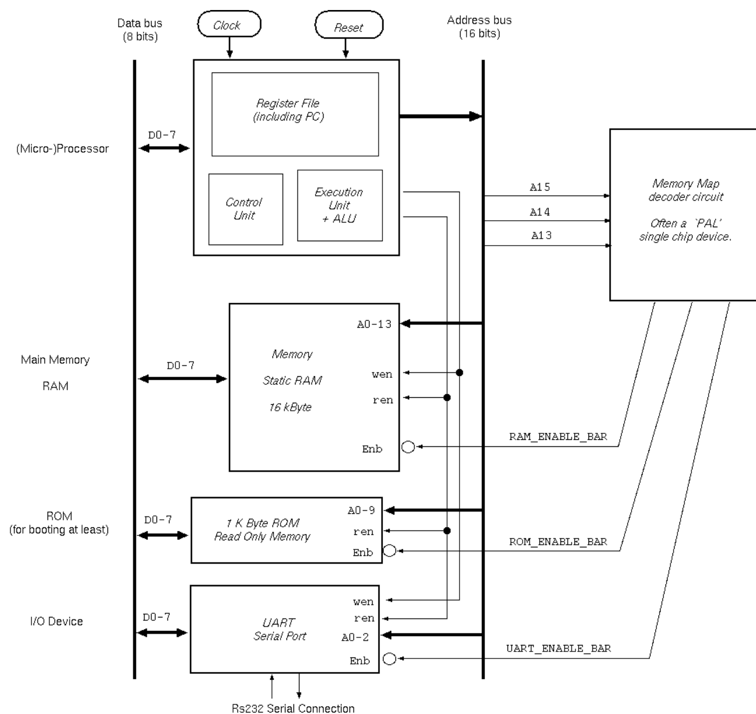


Figure 4.5: Early microcomputer structure, using a bidirectional/tri-state data bus.

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

4.2.2 Basic Bus: One initiator (II).

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

Figure 4.6 shows such a bus with one initiator and three targets.

No tri-states are used: on a modern SoC address and write data outputs use wire joints or buffers, read data uses multiplexors.

Max throughput is unity (i.e. one word per clock tick). Typical SoC bus capacity: 32 bits $\times$ 200 MHz = 6.4 Gb/s, but owing to protocol degrades with distance. This figure can be thought of as unity (i.e. one word per clock tick) in comparisons with other configurations we shall consider.

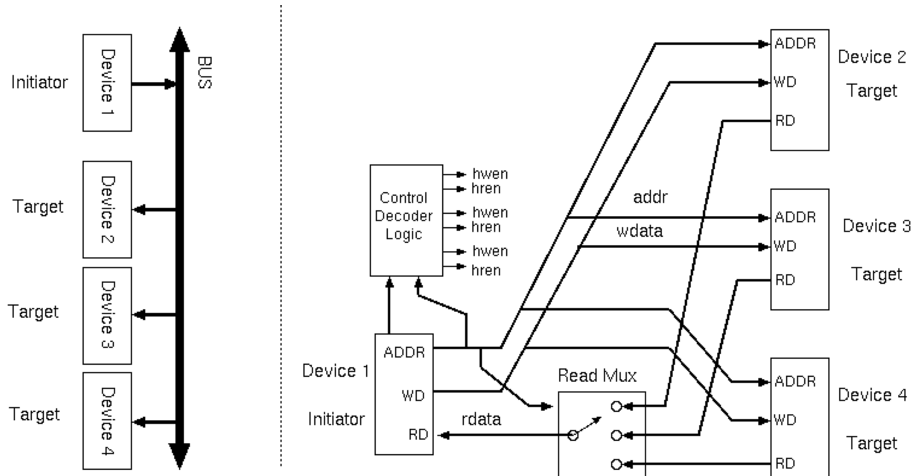


Figure 4.6: Example where one initiator addresses three targets.

The most basic bus has one initiator and several targets. The initiator does not need to arbitrate for the bus since it has no competitors.

Bus operations are reads or writes. In reality, on-chip busses support burst transactions, whereby multiple consecutive reads or writes can be performed as a single transaction with subsequent addresses being implied as offsets from the first address.

Interrupt signals are not shown in these figures. In a SoC they do not need to be part of the physical bus as such: they can just be dedicated wires running from device to device. (For ESL higher-level models and IP-XACT representation, interrupts need management in terms of allocation and naming in the same way as the data resources.)

Un-buffered wiring can potentially serve for the write and address busses, whereas multiplexors are needed for read data. Buffering is needed in all directions for busses that go a long way over the chip.

4.2.3 Basic bus: Multiple Initiators (II).

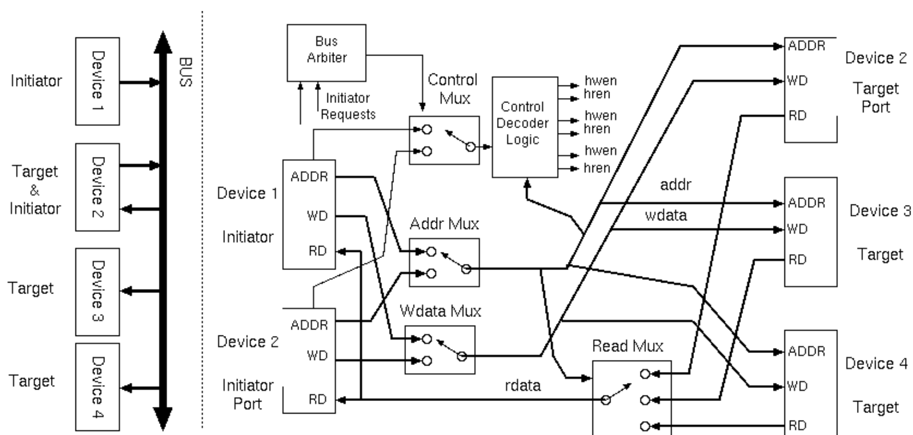


Figure 4.7: Example where one of the targets is also an initiator (e.g. a DMA controller).

Basic bus, but now with two initiating devices. Needs arbitration between initiators: static priority, round robin, etc.. With multiple initiators, the bus may be busy when a new initiator wants to use it, so there are various arbitration policies that might be used. Preemptive and non-preemptive with static priority, round robin and so on. The maximum bus throughput of unity is now shared among initiators.

Since cycles now take a variable time to complete, owing to contention, we certainly need acknowledge signals for each request and each operation (not shown).

How long to hold bus before re-arbitration ? Commonly re-arbitrate after every burst. The latency in a non-preemptive system depends on how long the bus is held for. Maximum bus holding times affect response times for urgent and real-time requirements.

4.2.4 Bridged Bus Structures.

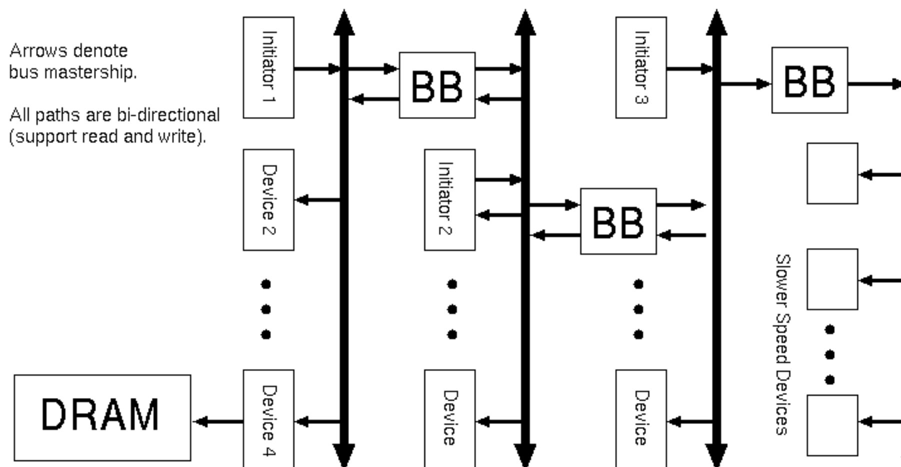


Figure 4.8: A system design using three main busses.

To make use of the additional capacity from bridged structures we need at least one main initiator for each bus. However, a low speed bus might not have its own initiators: it is just a slave to one of the other busses.

Bus bridges provide full or partial connectivity and some may write post. Global address space, non-uniform access time (NUMA). Some busses might be slower, narrower or in different clock domains from others.

The maximum throughput is the sum of that of all the busses that have their own initiators, but the achieved throughput will be lower if the bridges are used a lot: a bridged cycle consumes bandwidth on both sides.

How and where to connect DRAM is always a key design issue. The DRAM may be connected via a cache. The cache may be dual ported on to two busses, or more.

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

4.2.5 Classes of On-Chip Protocol

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

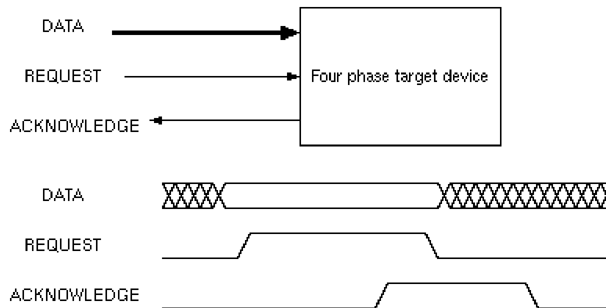


Figure 4.9: Timing diagram for an asynchronous, four-phase handshake.

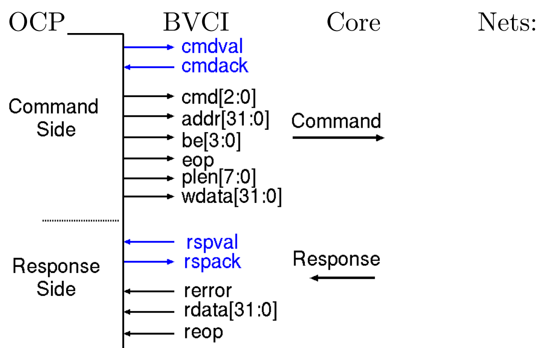
4. Virtual-circuit: (beyond scope of this course): rather than putting a destination address and port number in each message, traffic is routed at each hop via a short circuit identifier (or routing tag) where mappings have been set up in advance in the routing nodes.
5. Separated send and acknowledge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

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

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

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

Last year you used the Altera Avalon bus in part IB ECAD+Arch workshops. Many real-world IP blocks today are wired up using OCP's BVCI and ARM's AHB. Although the port on the IP block is fixed, in terms of its protocol, it can be connected to any system of bus bridges and on chip networks. Download full OCP documents from OCIP.org. See also bus-protocols-limit-design-reuse-of-ip



- All IP blocks can sport this interface.
- Separate request and response ports.
- Data is valid on overlap of `req` and `ack`.
- Temporal decoupling of directions:
- Allows pipeline delays for crossing switch fabrics or crossing clock domains.
- Sideband signals: interrupts, errors and resets: vary on per-block basis.
- Two complete instances of the port are neeed if block is both an initiator and target.
- Arrows indicate signal directions on initiator. All are reversed on target.

A prominent feature is totally separate request and response ports. This makes it highly tolerant of delays over the network and amenable to crossing clock domains. Older-style handshake protocols where

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

For each half of the port there are request and acknowledge signals, with data being transferred on any positive edge of the clock where both are asserted.

If a block is both an initiator and a target, such as our DMA controller example, then there are two complete instances of the port.

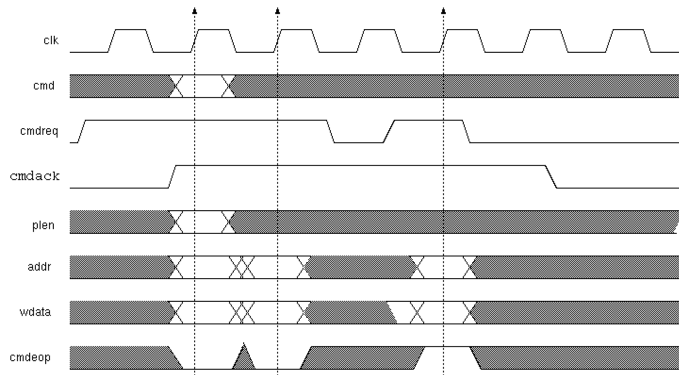
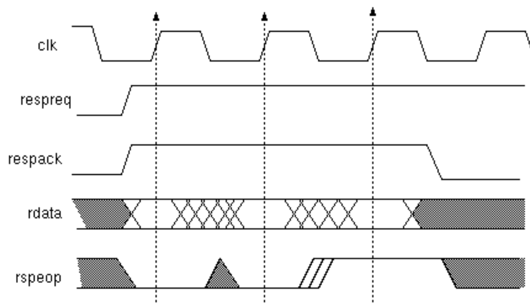


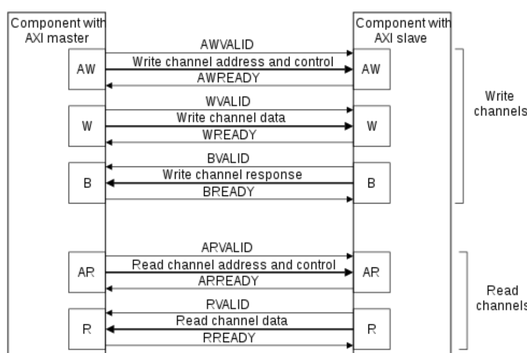
Figure 4.10: BVCI Protocol, Command Timing Diagram

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



BVCI Response Portion Protocol Timing Diagram

4.2.6 ARM AXI Bus: The Current Favourite

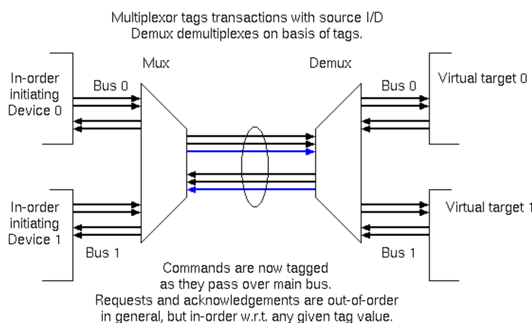


AXI-lite Bus Structure - Five temporally floating sub ports. One AXI port if formed of five separate interfaces that are called channels: two for read, three for write. Each of the five has its own contra-directional READY/VALID pair with all other nets running in the VALID direction and qualified by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

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

4.2.7 Supporting out-of-order operation using tags.

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



Out-of-order bus with tags.

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

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

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

4.3 RTL: Register Transfer Language

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

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

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

4.3.1 RTL Summary View of Variant Forms.

For the sake of this course, Verilog and VHDL are completely equivalent as register transfer languages (RTLs). Both support simulation and synthesis with nearly-identical paradigms. Of course, each has its proponent's.

Synthesisable Verilog constructs fall into these classes:

- **1. Structural RTL** enables an hierarchic component tree to be instantiated and supports wiring (a netlist) between components.
- **2. Lists of pure (unordered) register transfers** where the r.h.s. expressions describe potentially complex logic using a rich set of integer operators, including all those found in software languages such as C++ and Java. There is one list per synchronous clock domain. A list without a clock domain is for combinational logic (continuous assignments).
- **3. Synthesisable behavioural RTL** uses a thread to describe behaviour where a thread may write a variable more than once. A thread is introduced with the '`always`' keyword.

However, standards for synthesisable RTL greatly restrict the allowable patterns of execution: they do not allow a thread to leave the module where it was defined, they do not allow a variable to be written by more than one thread and they can restrict the amount of event control (i.e. waiting for clock edges) that the thread performs.

The remainder of the language contains the so-called 'non-synthesisable' constructs.

Additional notes:

The numerical value of any time values in RTL are ignored for synthesis. Components are synthesisable whether they have delays in them or not. For zero-delay components to be simulatable in a deterministic way the simulator core implements the **delta cycle** mechanism.

One can argue that anything written in RTL that describes deterministic and finite-state behaviour ought to be synthesisable. However, this is not what the community wanted in the past: they wanted a simple set of rules for generating hardware from RTL so that engineers could retain good control over circuit structures from what they wrote in the RTL.

Today, one might argue that the designer/programmer should not be forced into such low-level expression or into the excessively-parallel thought patterns that follow on. Certainly it is good that programmers are forced to express designs in ways that can be parallelised, but the tool chain perhaps should have much more control over the details of allocation of events to clock cycles and the state encoding.

RTL synthesis tools are not normally expected to re-time a design, or alter the amount of state or state encodings. Newer languages and flows (such as Bluespec and Kiwi) still encourage the user to express a design in parallel terms, yet provide easier to use constructs with the expectation that detailed timing and encoding might be chosen by the tool.

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

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

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

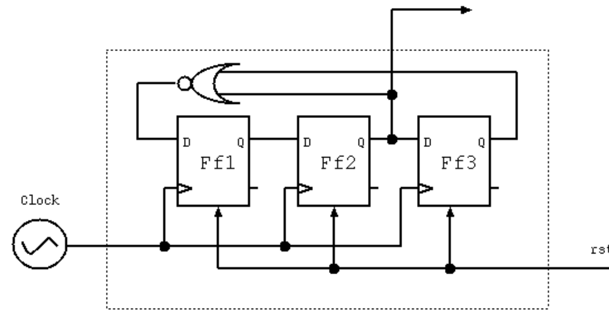


Figure 4.11: The circuit described by our structural example (a divide-by-five, synchronous counter).

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

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

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

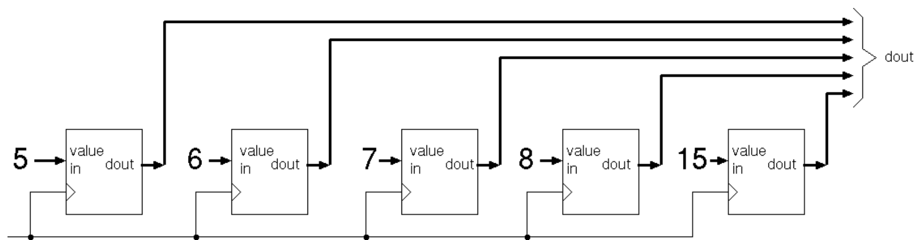
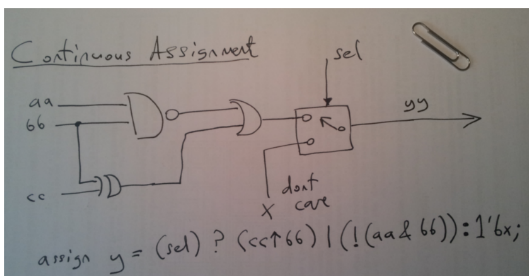


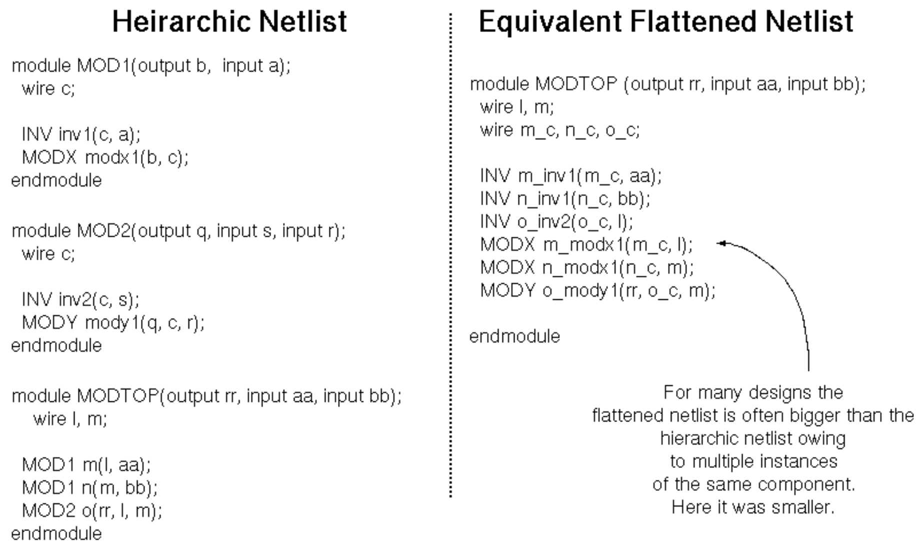
Figure 4.12: Example Generate Statement in RTL.

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

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




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



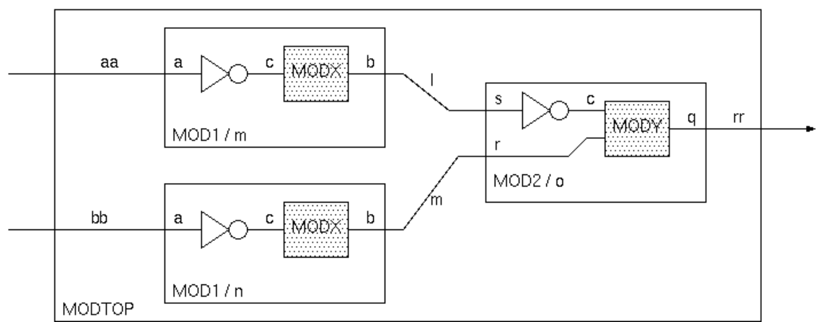


Figure 4.13: Example RTL fragment, before and after flattening.

- Order of continuous assignments is un-important,
- Tools often insist that continuous logic is loop-free, otherwise: intentional or un-intentional level-sensitive latches are formed (e.g. RS latch),
- Right-hand side's may range over rich operators (e.g. mux `?:` and multiply `*`),
- Bit inserts to vectors are allowed on left-hand sides (but not combinational array writes).

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

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

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

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

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

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

Typical example (illustrating pure RT forms):

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

      reg [3:0] count, oldcount;

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

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

   endmodule
```

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

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

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

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

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

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

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

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

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

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

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

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

Synthesis runs are made using points lower in the hierarchy as roots. We should certainly leave out the test-bench wrapper when synthesising and we typically want to synthesise each major component separately.

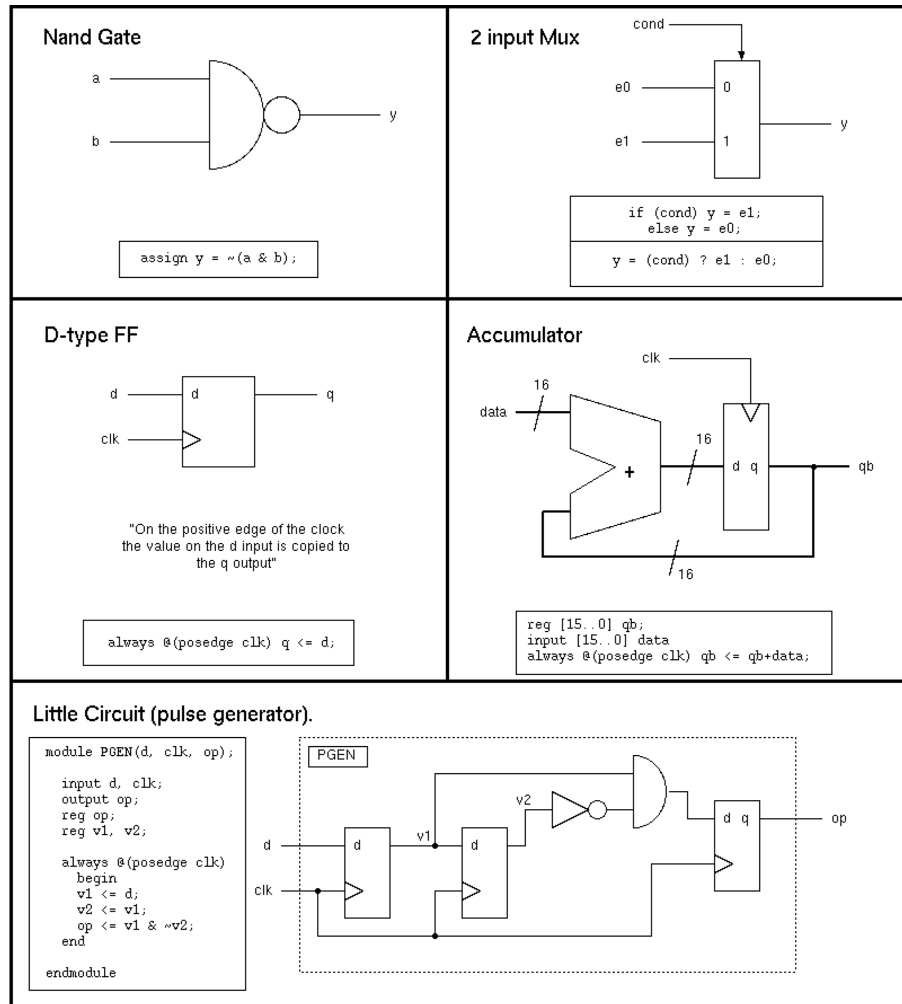


Figure 4.14: Elementary Synthesisable Verilog Constructs

4.3.2 Synthesisable RTL

Additional notes:

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

Expressions:

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

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

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

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

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

The abstract syntax tree for synthesisable RTL supports a rich set of expression operators but just the assignment and branching commands (no loops). (Loops in synthesisable VHDL and Verilog are restricted to the structural generation statements, mentioned above, that are fully unwound by the compiler front end and so have no data-dependent exit conditions).

An example of RTL synthesis:

Example input:

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

Results in structural RTL netlist:

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

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

Verilog RTL Synthesis Algorithm: 3-Step Recipe:

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

   For example:

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

   is converted to

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

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

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

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

is converted to

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

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

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

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

Further detail on selected constructs:

Additional notes:

1. How can we make a simple adder ?

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

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

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

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

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

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

4.3.3 Arrays and RAM Inference in RTL

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

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

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

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

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

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

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

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

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

4.3.4 Behavioural - 'Non-Synthesisable' RTL

Not all RTL is officially synthesisable, as defined by language standards. However, commercial tools tend to support larger subsets than officially standardised.

RTL with event control in the body of a thread defines a state machine. This is compilable by some tools. This state machine requires a program counter (PC) register at runtime (implied):

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

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

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

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

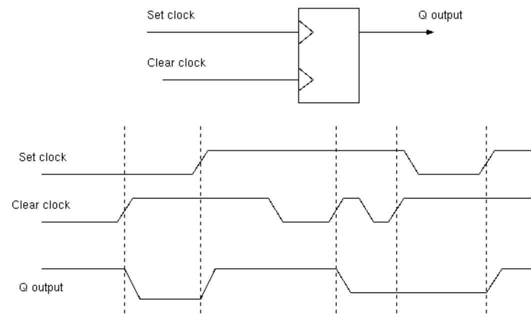


Figure 4.15: Schematic symbol and timing diagram for an edge-triggered RS flop.

```
reg q;
input set, clear;

always @(posedge set) q = 1;
always @(posedge clear) q = 0;
```

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

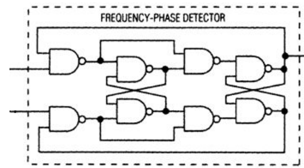


Figure 4.16: Hand-crafted circuit for the edge-triggered RS flop used in practice.

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

```
// Typical RTL testbench contents:

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

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

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

4.3.5 Further Logic Synthesis Issues

There are many combinational circuits that have the same functionality. Synthesis tools can accept additional guiding metrics from the user, that affect

- Power consumption (with automatic clock gating synthesis),
- Area use,

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

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

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

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

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

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

4.4 Simulation

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

There are two main forms of simulation modelling:

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

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

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

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

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

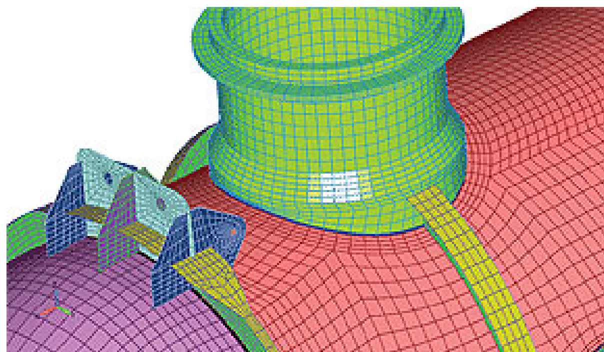


Figure 4.17: Finite Element Grid

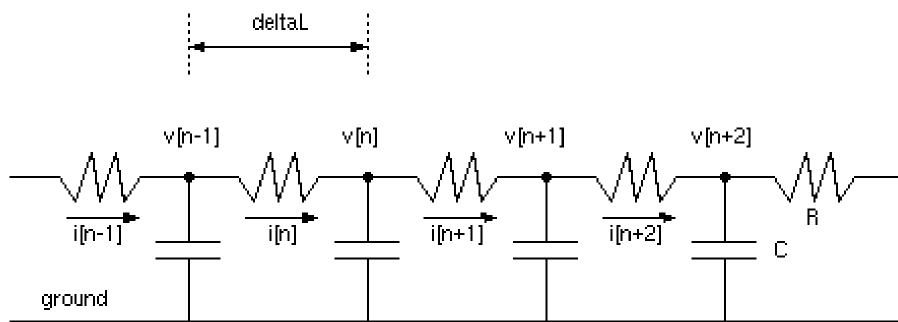


Figure 4.18: Baseline finite-difference model for bidirectional propagation in one dimension.

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

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

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

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

4.4.1 Digital Logic Modelling

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

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

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

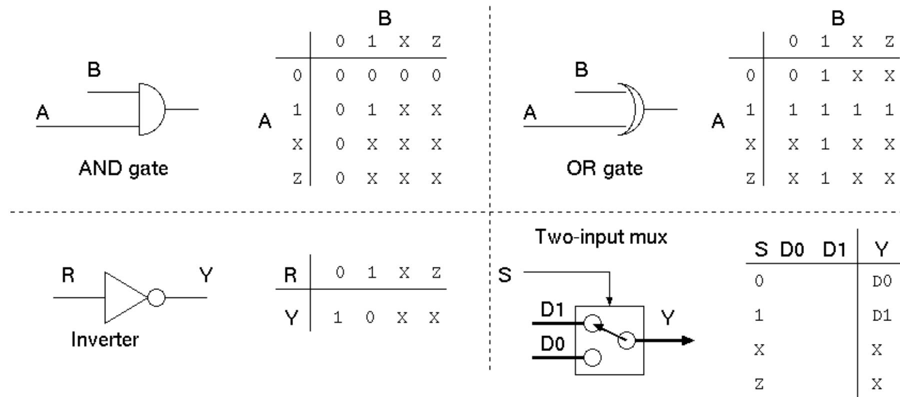


Figure 4.19: Illustratring the four-value logic level encoding for common gates.

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

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

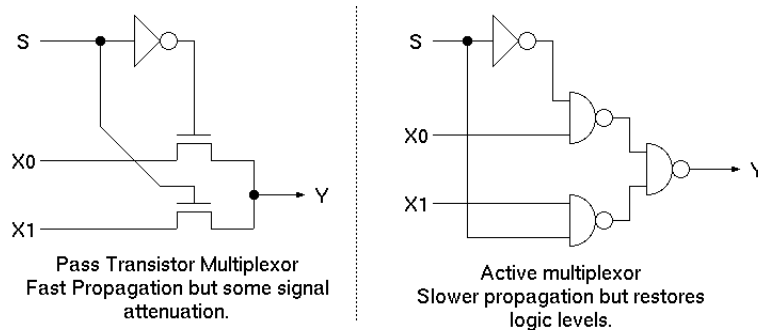


Figure 4.20: Pass transistor multiplexor compared with an active path multiplexor that needs more silicon area, especially when the control inverter is amortised over a word.

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

4.4.2 Event Driven Simulation

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

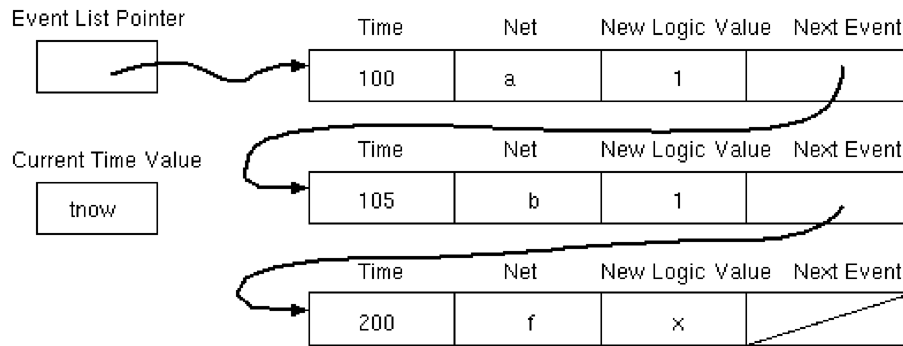


Figure 4.21: Event queue, linked list, sorted in ascending temporal order.

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

type value_t = V_n of int | V_z | V_x;

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

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

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

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

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

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

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

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

Main simulation: keep dispatching until event list empty:

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

4.4.3 Mixed Analog/Digital Simulation (Verilog-AMS)

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

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

Example: Vehicle hybrid power system and automatic braking system.

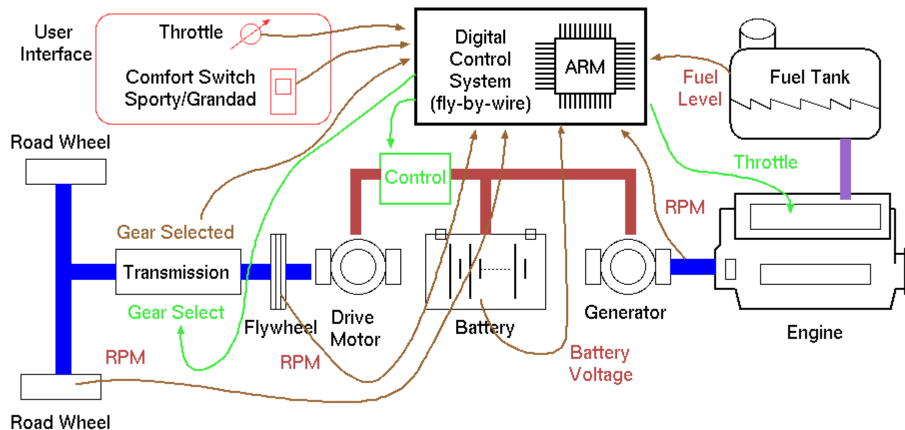


Figure 4.22: Hybrid Automobile Transmission System.

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

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

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

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

Examples:

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

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

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

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

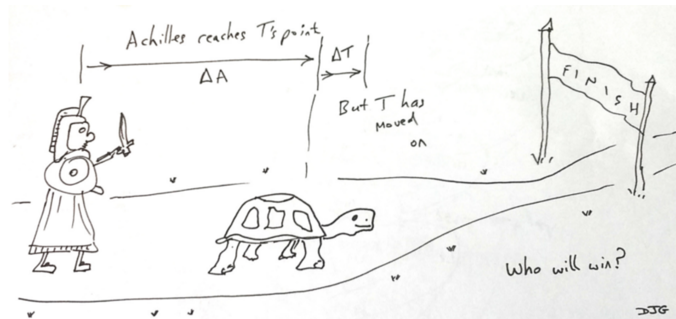


Figure 4.23: Archilles can never catch the tortoise? Sum of GP never converges?

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

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

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

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

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

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

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

4.4.5 Higher-level Simulation

Simulating RTL is slow. Every net (wire) in the design is modelled as a shared variable. When one component writes a value, the overheads of waking up the receiving component(s) may be severe. The

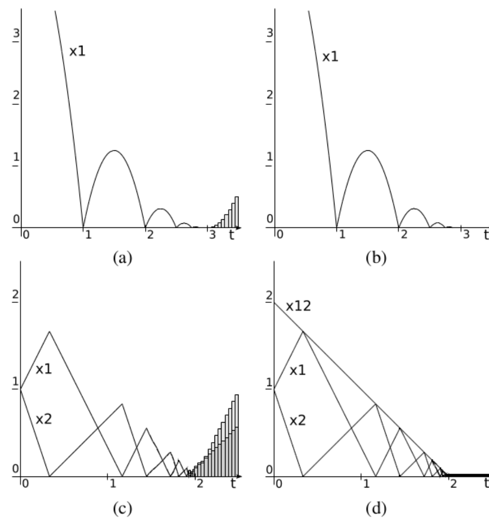


Fig. 2: Enclosures for systems with Zeno behavior. Figures (a) and (c) show the results for the bouncing ball and water tank systems of Lygeros [33, Figures 3.1 and 3.7]. However, we make these models more challenging to simulate by replacing some inequalities with equalities (such as the bouncing conditions).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

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

event-driven simulator kernel contains an indirect jump instruction which itself is very slow on modern computer architectures since it will not get predicted correctly.

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

4.5 Hazards

Definitions (some authors vary slightly):

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

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

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

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

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

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

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

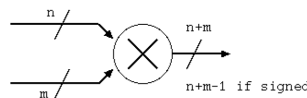
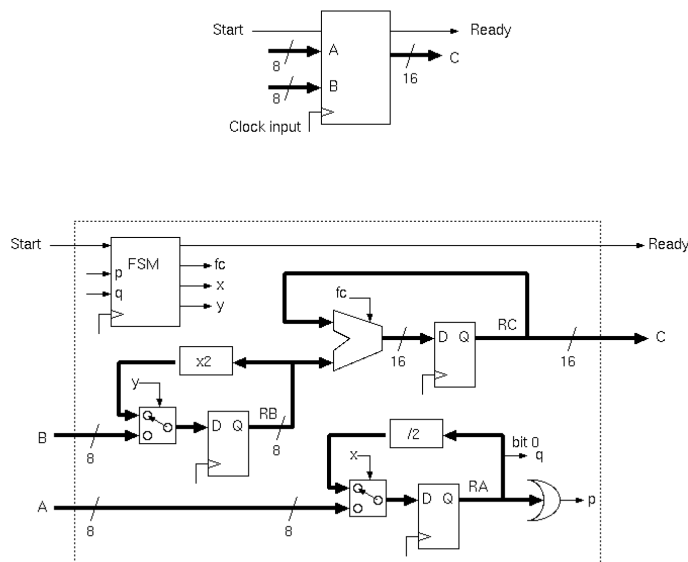


Figure 4.25: Multiplier schematic symbol.

Behavioural algorithm:

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

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



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

4.5.1 Hazards From Array Memories

A structural hazard in an RTL design can make it non synthesisable. Consider the following expressions that make liberal use of array subscription and the multiplier operator:

Structural hazard sources are numbered:

```
always @(posedge clk) begin
   q0 <= Boz[e3]             // 3
   q1 <= Foo[e0] + Foo[e1]; // 1
   q2 <= Bar[Bar[e2]];      // 2
   q3 <= a*b + c*d;         // 4
   q4 <= Boz[e4]            // 3
   end
```

1. The RAMs or register files Foo Bar and Boz might not have two read ports.
2. Even with two ports, can Bar perform the double subscription in one clock cycle?
3. Read operations on Boz might be a long way apart in the code, so hazard is hard to spot.
4. The cost of providing two 'flash' multipliers for use in one clock cycle while they lie idle much of the rest of the time is likely not warranted.

A multiplier that operates combinationaly in less than one clock cycle is called a 'flash' multiplier and it uses quadratic silicon area.

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

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

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

Multipliers and floating point units also typically present hazards.

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

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

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

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

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

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

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

4.5.2 Overcoming Structural Hazards using Holding Registers

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

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

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

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

then load holding register in additional cycle:

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

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

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

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

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

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

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

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

The transformations illustrated above are NOT performed by mainstream RTL compilers today: instead they are incorporated in HLS tools such as Kiwi. KiwiC Structural Hazard ExampleSharing structural resources may require additional multiplexers and wiring: so not always worth it. A good design not only balances structural resource use between clock cycles, but also critical path timing delays.

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

4.6 Folding, Retiming & Recoding

Generally we have to chose between high performance or low power. (We can see this also in the selection of drive strengths for standard cell gates.) The **time/space fold** and **unfold** operations trade execution time for silcon area. A given function can be computed with fewer clocks by 'unfolding' in the the time domain, typically by loop unwinding (and predication).

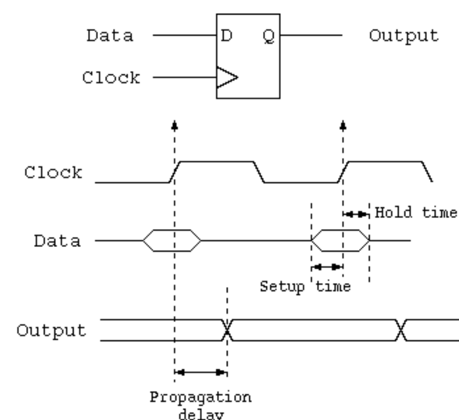
```
LOOPED (time) option:                  | UNWOUND (space) option:
                                       |
for (i=0; i < 3 and i < limit; i++)    |  if (0 < limit) sum += data[0] * coef[j];
   sum += data[i] * coef[i+j];         |  if (1 < limit) sum += data[1] * coef[1+j];
                      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

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

We can **retime** a design with and without changing its state encoding. We will see that adding a pipeline stage can increase the amount of state without **recoding** existing state. *Note: these folding operations, discussed here in their RTL form, are precursors to considering their automated deployment by HLS tools.*

4.6.1 Critical Path Timing Delay

Meeting **timing closure** is the process of manipulating a design to meet its target clock rate (as set by the Marketing Department for instance). *We reviewed this figure earlier in the course.*



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

Pipelining is a commonly-used technique to boost system performance. Introducing a pipeline stage increases latency but also the maximum clock frequency. Fortunately, many applications are tolerant to the processing delay of a logic subsystem. Consider a decoder for a fibre optic signal: the fibre might be many kilometers long and a few additional clock cycles in the decoder increase the processing delay by an amount equivalent to a few coding symbol wavelengths: e.g. 20 cm per pipeline stage for a 1 Gbaud modulation.

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

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

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

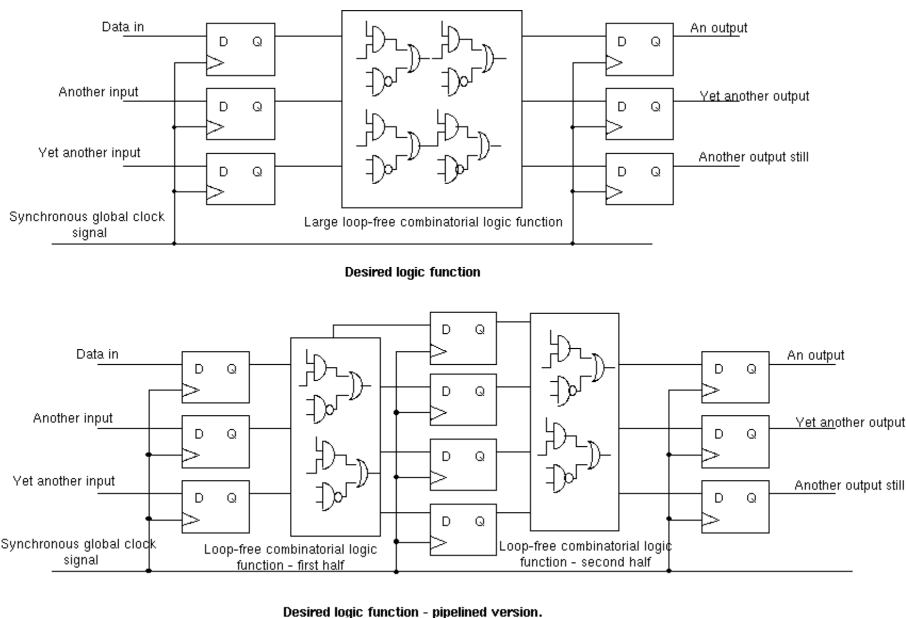


Figure 4.26: A circuit before and after insertion of an additional pipeline stage.

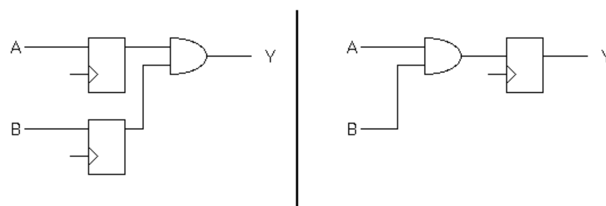


Figure 4.27: Flip-flop migration: two circuits of identical behaviour, but different state encoding.

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

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

Problems arising:

- Circuits containing loops (proper synchronous loops) cannot be pushed very far (for example, the control hazard in a RISC pipeline).
- External interfaces that do not use transactional handshakes (i.e. those without flow control) cannot tolerate automatic re-timing since the knowledge about when data is valid is not explicit.
- Many structures, including RAMs and ALUs, have a pipeline delay (or several), so the hazard on their input port needs resolving in a different clock cycle from hazards involving their result values.

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

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

4.6.2 Static Timing Analyser Tool

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

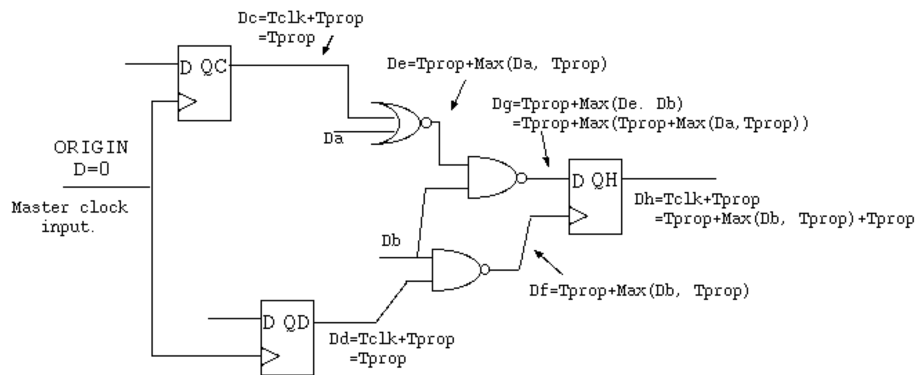


Figure 4.28: An example circuit with static timing annotations

A static timing analyser computes the longest event path through logic gates and clock-to-Q paths of edge-triggered flops. The longest path is generally the critical path that sets the maximum clock frequency. However, sometimes this is a false result, since this path might never be used during device operation.

Starting with some reference point, taken as D=0, such as the master clock input to a clock domain, we compute the relative delay on the output of each gate and flop. For a combinational gate, the output delay is the gate's propagation time plus the maximum of its input delays. For an edge-triggered flop, such as a D-type or a JK, there is no event path to the output from the D or JK inputs, so it is just the clock delay plus the flop's clock-to-Q delay. There are event paths from asynchronous flop inputs however, such as preset, reset or transparent latch inputs.

Propagation delays may not be the same for all inputs to a given output and for all directions of transition. For instance, on deassert of asynchronous preset to a flop there is no event path. Therefore, a tool may typically keep separate track of high-to-low and low-to-high delays.

4.6.3 Back Annotation and Timing Closure

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

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

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

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

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

4.6.4 Conventional RTL Compared with Software

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

RTL is statically-allocated and defines a finite-state machine. Threads do not leave their starting context

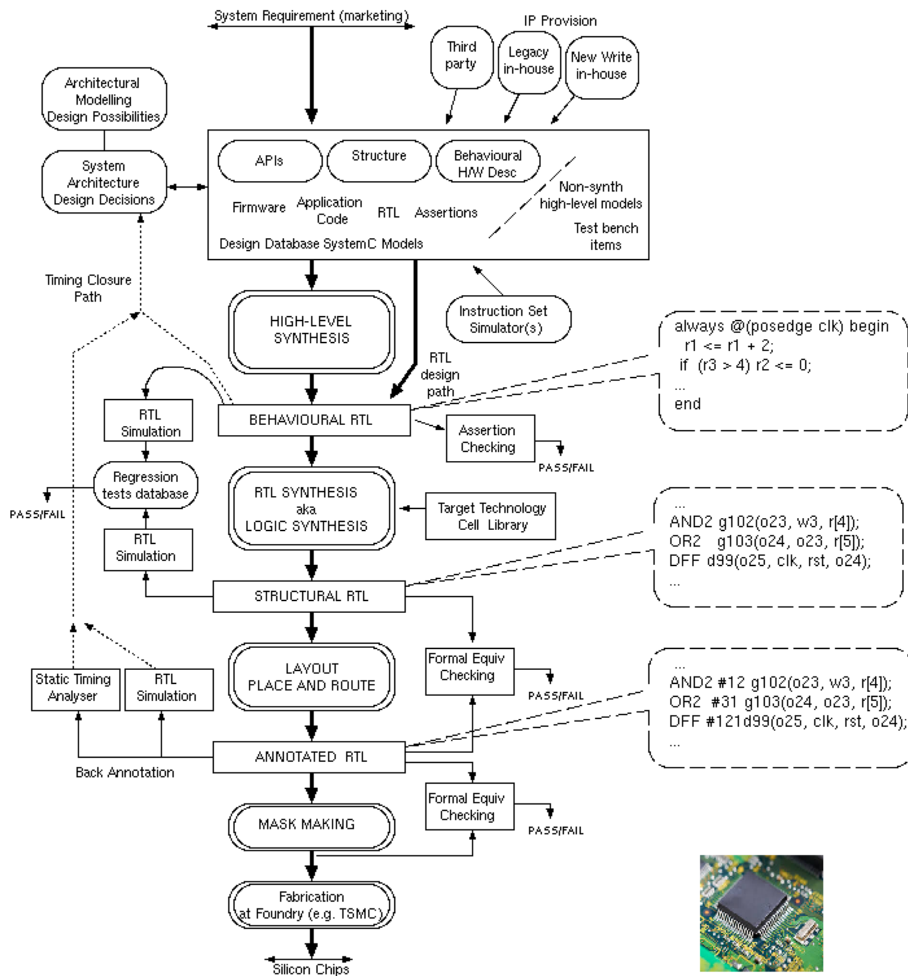


Figure 4.29: Design and Manufacturing Flow for SoC.

and all communication is through shared variables that denote wires. There are no thread synchronisation primitives, except to wait on a clock edge. Each variable must be updated by at most one thread.

Software on the other hand uses far fewer threads: just where needed. The threads may pass from one module to another and thread blocking is used for flow control of the data. RTL requires the programmer to think in a massively-parallel way and leaves no freedom for the execution platform to reschedule the design.

RTL is not as expressive for algorithms or data structures as most software programming languages.

The concurrency model is that everything executes in lock-step. The programmer keeps all this concurrency in his/her mind. Users must generate their own, bespoke handshaking and flow control between components.

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

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

KG 5 — High-level Design Capture and Synthesis

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

5.0.5 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.

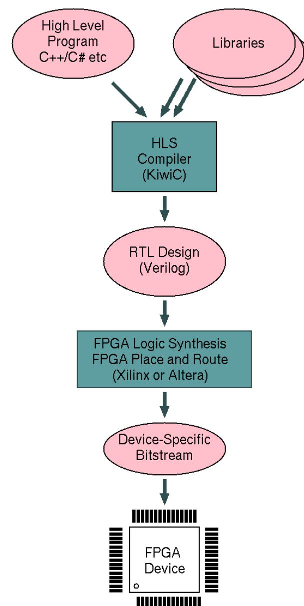


Figure 5.1: Basic Steps of an HLS Flow.

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

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

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

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

Traditional RTL design entry (Verilog/VHDL) needs:

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

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

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

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

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

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

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

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

5.0.6 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 §1.3.2). Three forms of parallel speed-up are well-known for classical imperative parallel programming:

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

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

5.0.7 Instruction-Level Parallelism

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

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

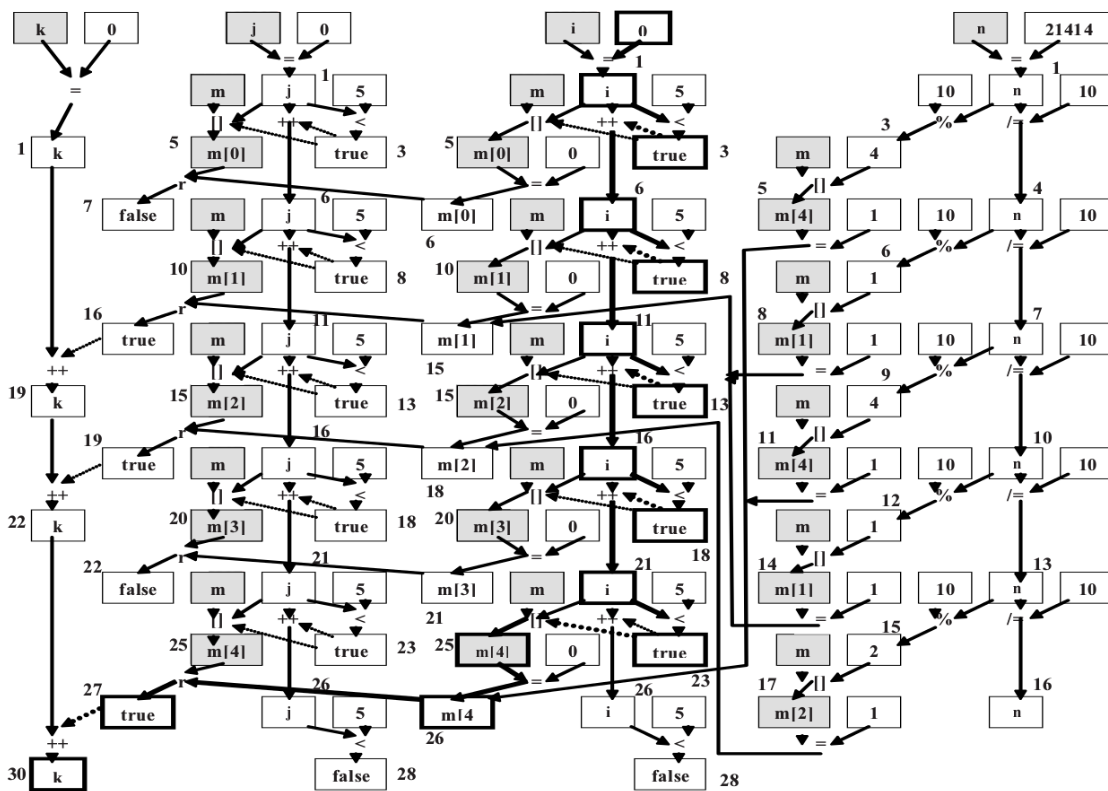


Figure 5.2: Critical Path in Digit Counting C Program (Prihozhy).

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

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

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

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

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

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

(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.)

5.0.8 Beyond Pure RTL: Behavioural descriptions of hardware.

What has 'synthesisable' RTL traditionally provided ?

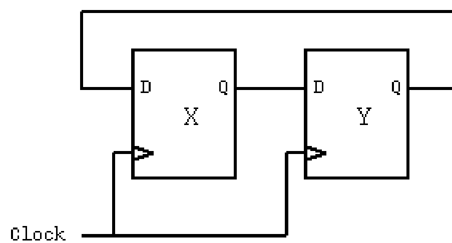


Figure 5.3: A circuit to swap two registers.

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

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

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

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

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

With this style of specification the order of the statements is significant and typically such assignment statements are incorporated in various nested `if-then-else` and `case` commands. This allows hardware designs to be expressed using the conventional imperative programming style that is familiar to software programmers. The intention of this style is to give an easy to write and understand description of the desired function, but this can result in logic output from the synthesiser which is mostly incomprehensible if inspected by hand.

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

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

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

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

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

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

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

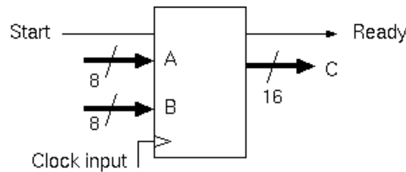
```
if (din) @(posedge clk) yout = 4;
```

Take a non-rentrant function:

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

that achieves the same behaviour.

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



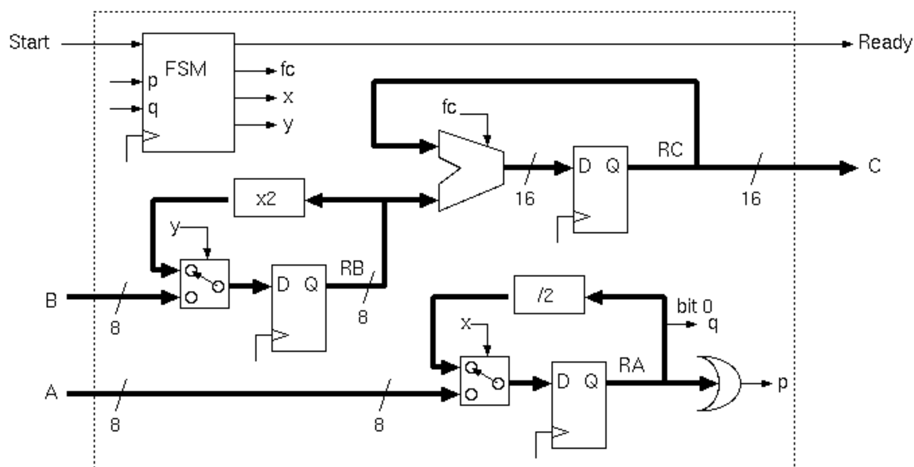


Figure 5.4: Long multiplier viewed as datapath and sequencer.

This simple example has no multi-cycle primitives and a 1-to-1 mapping of ALUs to the source code text, so no schedulling was needed from the HLS tool. Each register has a multiplexor that ranges over all the places it is loaded from. We followed a **syntax-directed** approach (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.

5.0.9 Classical HLS Compiler: Operational Phases

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

1. **Lexing and Parsing** as for any HLL
2. **Type and reference checking**: can an int be added to a string? Is an invoked primitive supported?
3. **Trimming**: Unreachable code is deleted, register widths are reduced where it is manifest that the value stored is bounded, constants are propagated between code blocks and identity reductions are applied to operators, such as multiplying by unity.

4. **Binding**: Every storage and processing element, such as a variable or an add operation or memory read, is allocated a physical resource.
5. **Polyhedral Mapping**: A memory layout or ordering optimisation for nested loops.
6. **Schedulling**: Each physical resource will be used many times over in the time domain. A static schedule is generated. This is typically a scoreboard of what expressions are available when. (Worked example in lecuters.)
7. **Sequencer Generation**: A controlling FSM that embodies the schedule and drives multiplexor and ALU function codes is generated.
8. **Quantity Surveying**: The number of hardware resources and clock cycles used can now be readily computed.
9. **Optimisation**: The binding and schedulling phase may be revisited to better match user-provided target metrics.
10. **RTL output**: The resulting design is printed to a Verilog or VHDL file.

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

Important binding decisions arise for memories:

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

5.0.10 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).

5.0.11 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).

5.0.12 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.

5.0.13 Pipelined Schedulling - One Basic Block

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

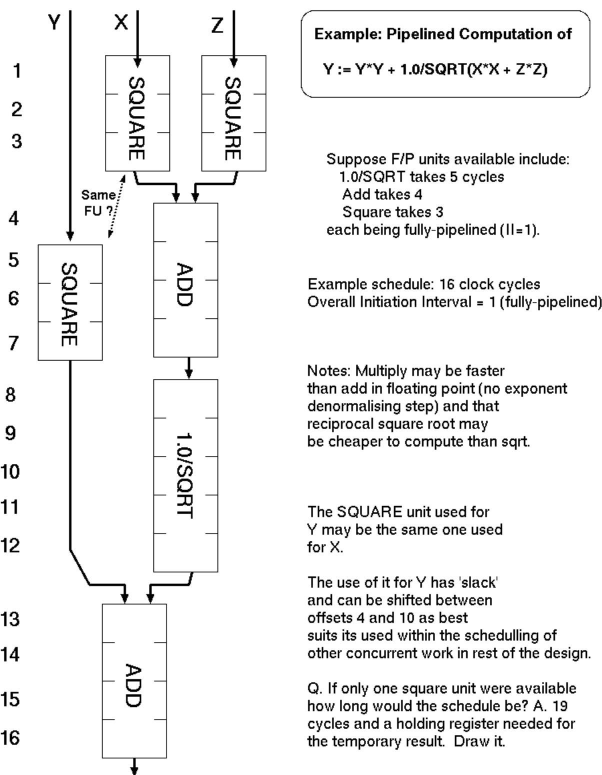


Figure 5.5: Example static schedule for a basic block containing a single assignment.

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

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

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

5.0.14 Pipelined Schedulling - Between Basic Blocks

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

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

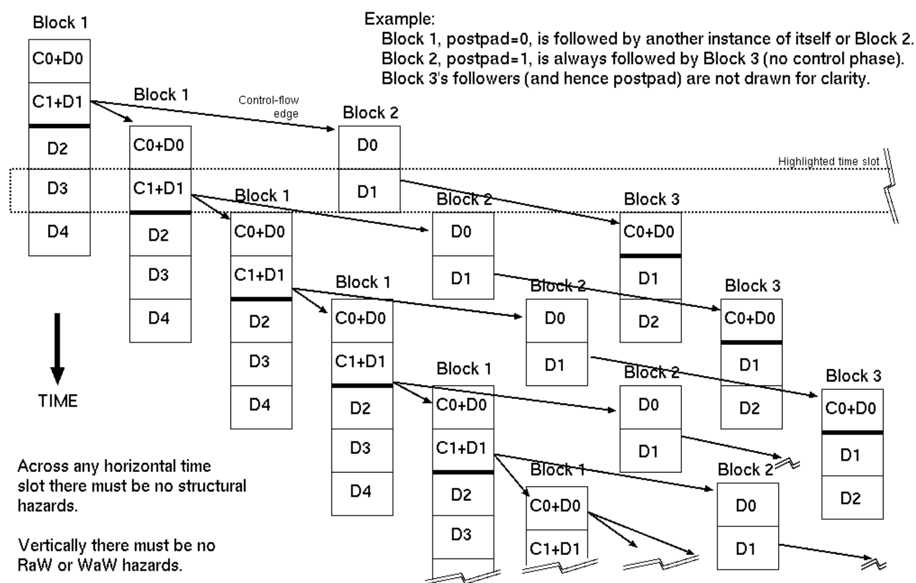


Figure 5.6: Fragment of an example inter-block initiation and hazard graph.

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 5.6, the D3 operations of the first block are concurrent with the

control and data C1+D1 operations of a later copy of itself when it has looped back or with the D1 phase of Block 2 if exited from its tight loop.

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

5.1 HLS Functional Units (FUs)

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

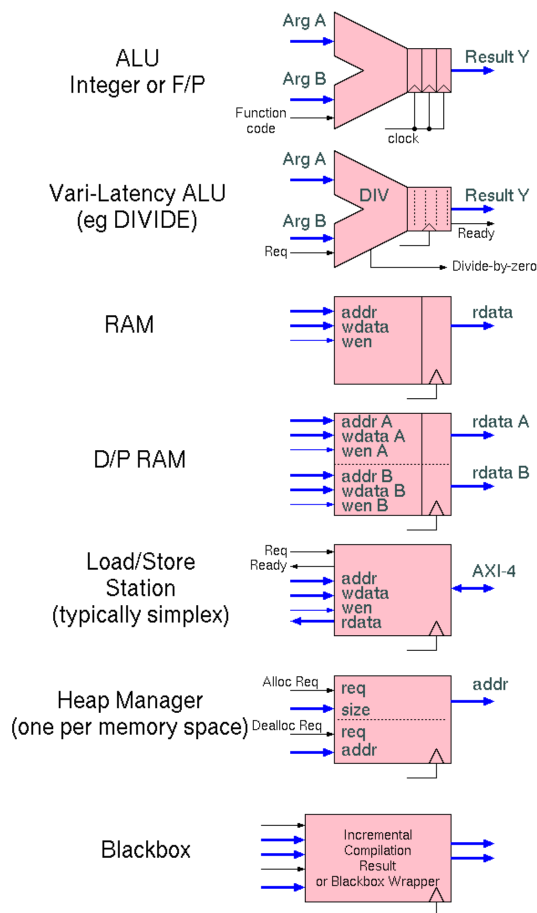


Figure 5.7: Typical examples of FUs deployed by an HLS compiler.

5.1.1 Functional Unit (FU) Block Properties

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

- int **Precision**
- bool **Referentially-Transparent (Stateless)**: always same result for same arguments.

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

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

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

5.1.2 Functional Unit (FU) Chaining

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

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

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

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

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

5.2 Discovering Parallelism: Classical HLS Paradigms

The well-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.

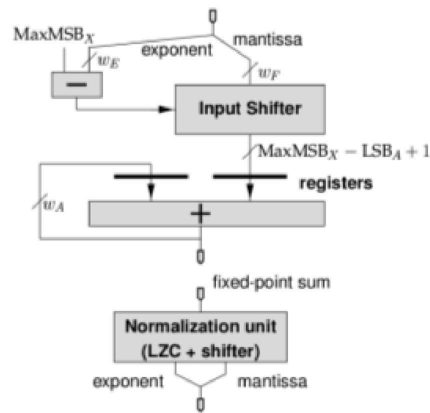


Figure 5: The proposed accumulator. Only the registers on the accumulator itself are shown, the rest of the design is combinatorial and can be pipelined arbitrarily.

Figure 5.8: Fixed-Point Accumulator Proposal.

```
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 5.5). When speculating we continue looping but provide a mechanism to discard unwanted side effects.

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

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

5.2.1 Memory Banking and Widening

Whether computing on standard CPUs or FPGA, memory bandwidth is often 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.

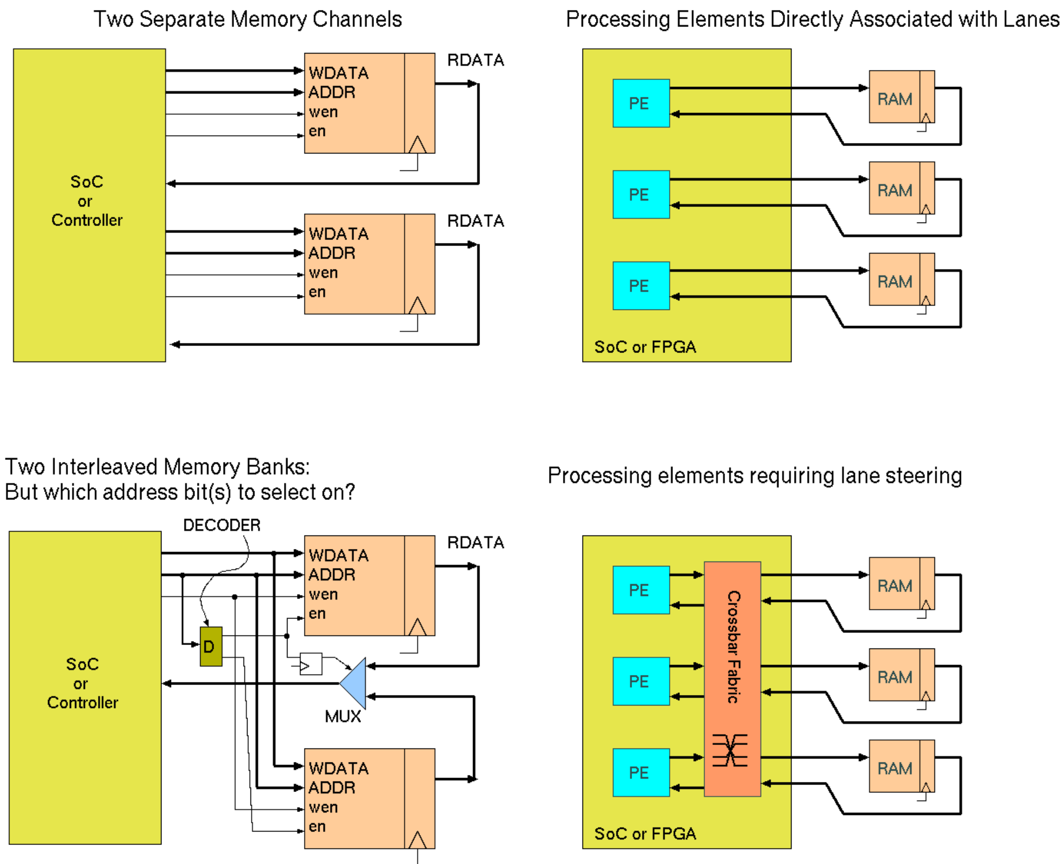


Figure 5.9: Alternative Memory Bank Structures giving wider effective bus width.

5.2.2 Data Layout for Burrows-Wheeler Transform

The BWT of a string is another string of the same length and alphabet. According to the definition of a transform, it is information preserving and has an inverse. We shall not define it in these notes. The following code makes efficient perfect string matches of needles in a given haystack using the BWT. It uses lookup in the large Rank array that is pre-computed from 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 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

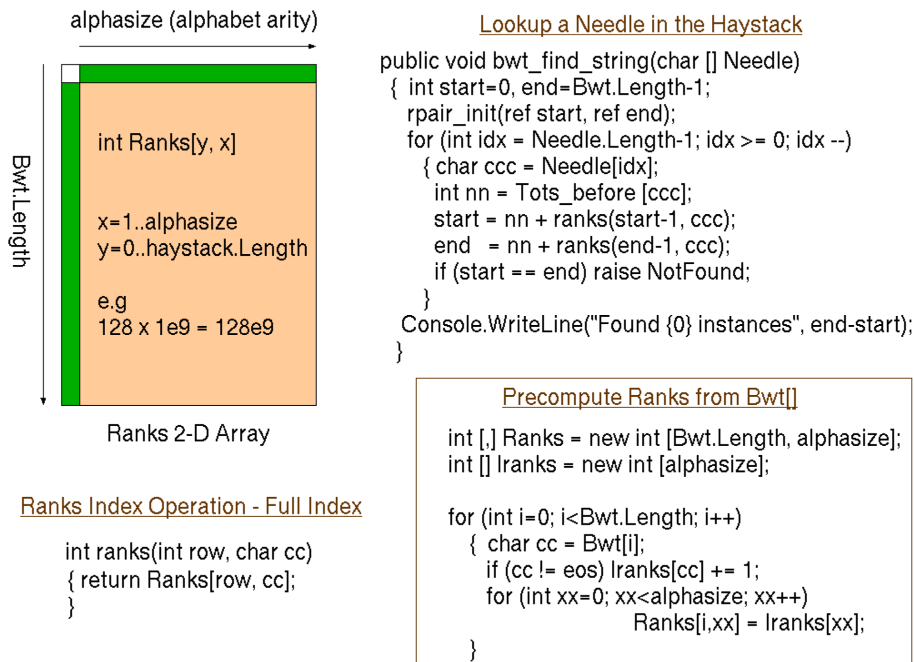


Figure 5.10: Lookup procedure when string searching using BWT.

with the good alignment.

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

5.2.3 Smith-Waterman D/P Data Dependencies

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

A quadratic algorithm based on dynamic programming is used. The maximum score needs to be found in a 2-D array where each score depends on the three immediate neighbours with lower index as shown in figure 5.12. Zeros are inserted where subscripts would be negative at the edges.

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

We shall discuss suitable hardware architectures on the example sheet.

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

5.2.4 Polyhedral Address Mapping

Polyhedral Address Mapping not examinable for CST Part II.

A number of restricted mathematical systems have useful results in terms of decidibility of certain propositions. A restriction might be that expressions only multiply where one operand is a constant and a

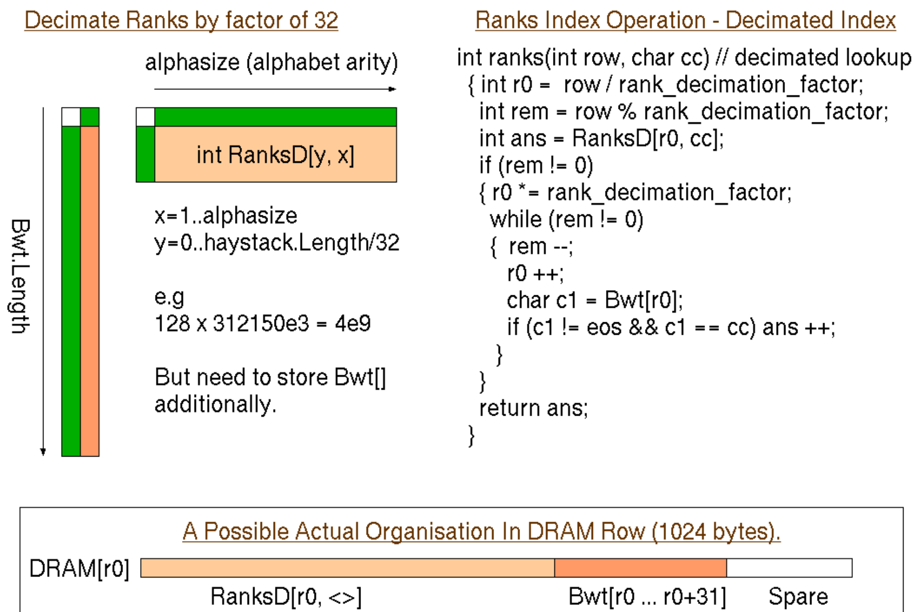


Figure 5.11: Compacted Rank Array for BWT and a sensible layout in a DRAM row.

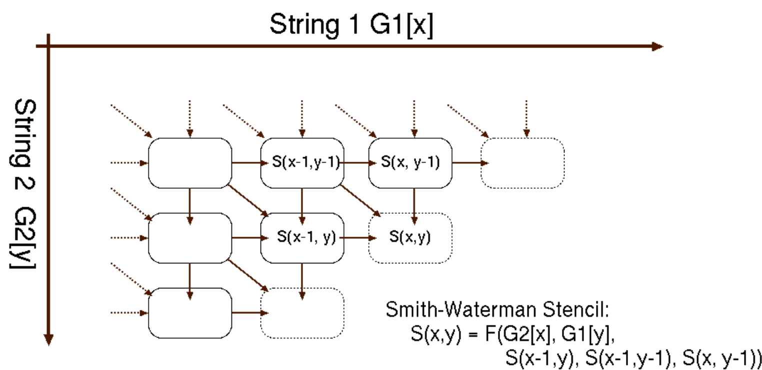


Figure 5.12: Data dependencies (slightly simplified) in Smith-Waterman Alignment Matcher.

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 or polytope space. This space is scanned by the vector consisting of the induction variables.

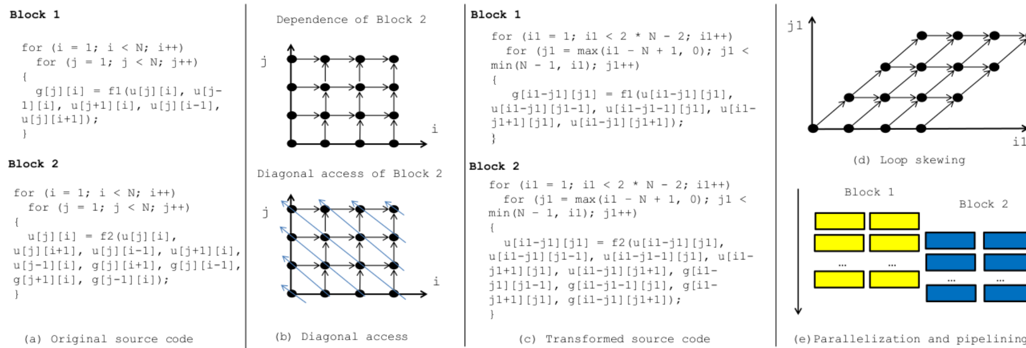


Figure 5.13: Affine transformations (Zuo, Liang et al).

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 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.”

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?

5.2.5 The Perfect Shuffle Network - FFT Example

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

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

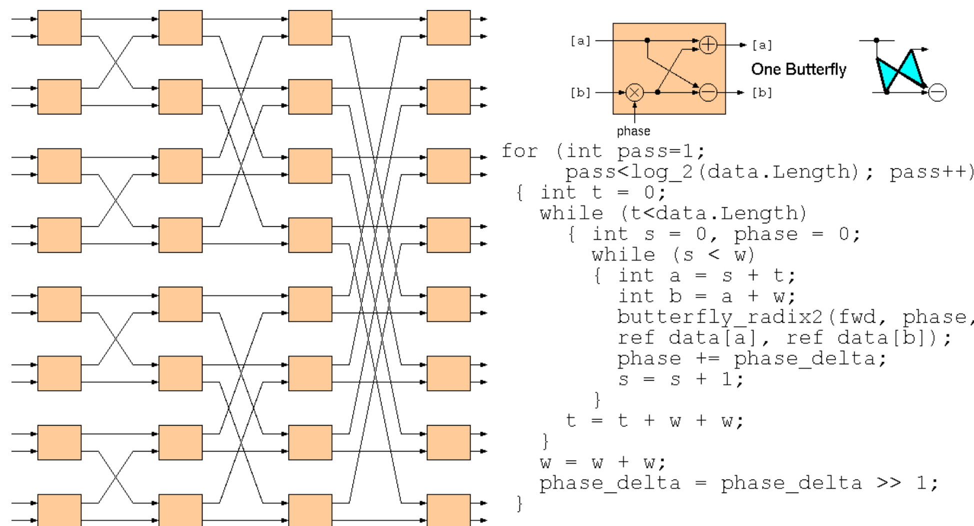


Figure 5.14: Shuffle Data Flow in the Fast Fourier Transform.

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.

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

5.2.6 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 multi-processors (CMPs),
- is generally better avoided (so say many at least)!

Kahn Networks and Systolic Arrays are not examinable in 2017/18 SoC D&M.

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 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?

5.2.7 Other Expression forms: Hardware Construction Languages

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

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

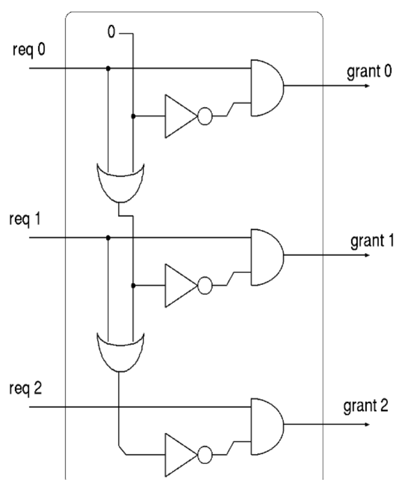
Using guarded atomic actions is an old and well-loved design paradigm. Recently Bluespec System Verilog has successfully raised the level of abstraction in RTL design using this paradigm.

- Every leaf operation has a guard predicate: says when it CAN be run.
- A Bluespec design is expressed as a list of 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.

A Varadic Priority Arbiter in Chisel

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

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



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

David J Greaves – Computer Lab Cambridge

Memocode 2015, Austin Texas.

Figure 5.15: An Example Chisel Module.

- 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
```

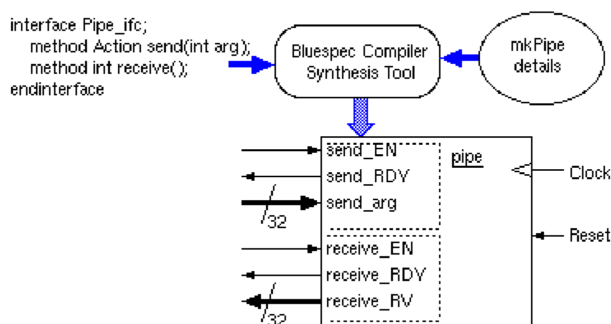


Figure 5.16: Synthesis of the ‘pipe’ Bluespec component with handshake nets.

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.

5.2.9 Classical Imperative/Behavioural H/L Synthesis Summary

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

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

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

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

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

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

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

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

5.2.10 Accellera IP-XACT

IP-XACT non-examinable for Part II CST.

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

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

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;

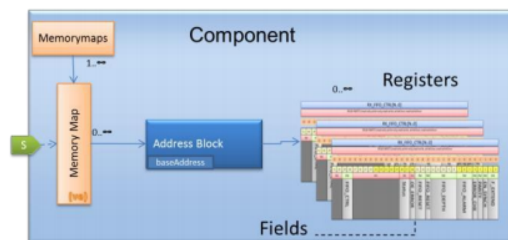


Figure 5.17: IP-XACT captures memory map and register field definitions.

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

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

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

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

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

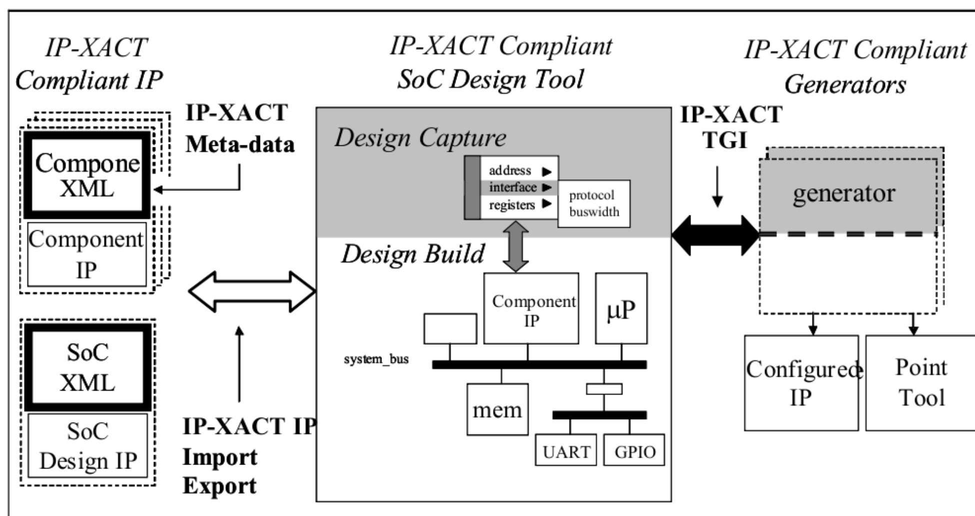


Figure 5.18: Reference Model for design capture and synthesis using IP-XACT blocks.

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

Automatic generation of memory maps and device-driver header files is also normally supported. Header

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

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

KG 6 — Architectural Exploration using ESL Modelling

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

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

An ESL methodology aims:

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

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

So, an ESL methodology must provide:

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

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

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

6.0.11 SystemC: Hardware Modelling Library Overview

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

It includes (at least):

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

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

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

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

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

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

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

Additional notes:

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

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

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

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

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

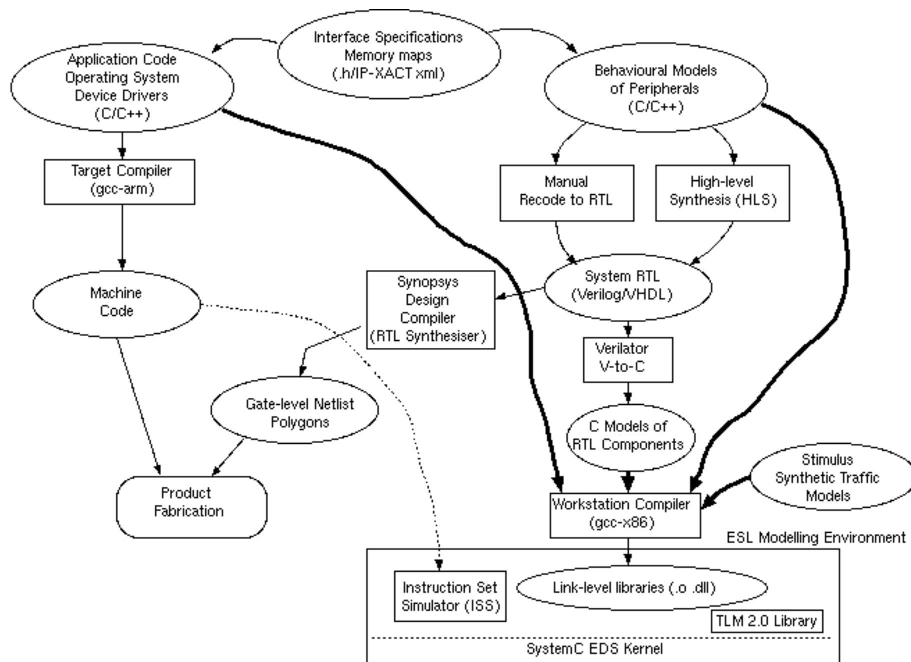


Figure 6.1: ESL Flow: Avoiding the ISS by cross-compiling the firmware and direct linking with behavioural models.

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

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

6.1.1 Using C Preprocessor to Adapt Firmware

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

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

#ifdef ACTUAL_FIRMWARE

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

#else

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

#endif

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

The remainder of this slide/section links to details of a specific example not lectured in 2017/18.

DMA Controller RTL Version (from 2016 SoC Parts Slide Pack)

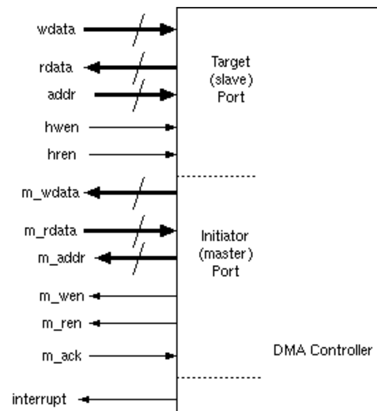


Figure 6.2: Block Diagram of one-channel DMA unit.

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

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

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

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

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

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

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

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

6.2 Transactional-Level Modelling (TLM)

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

1. **Pin-level modelling**: an event is a change of a single-bit or multi-bit net,
2. **Abstract data modelling**: an event is delivery of a large data packet, such as a complete cache line,
3. **Transactional-level modelling**: avoid events as much as possible: use intermodule software calling.

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

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

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

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

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

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

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

Figure 6.3 is an example protocol implemented at net-level and TLM level:

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

6.2.1 General ESL Interactions with Shortcuts Illustrated

This slide/section not lectured in 2017/18. But you may like to look at the Ethernet CRC Example Consider the Ethernet CRC Example

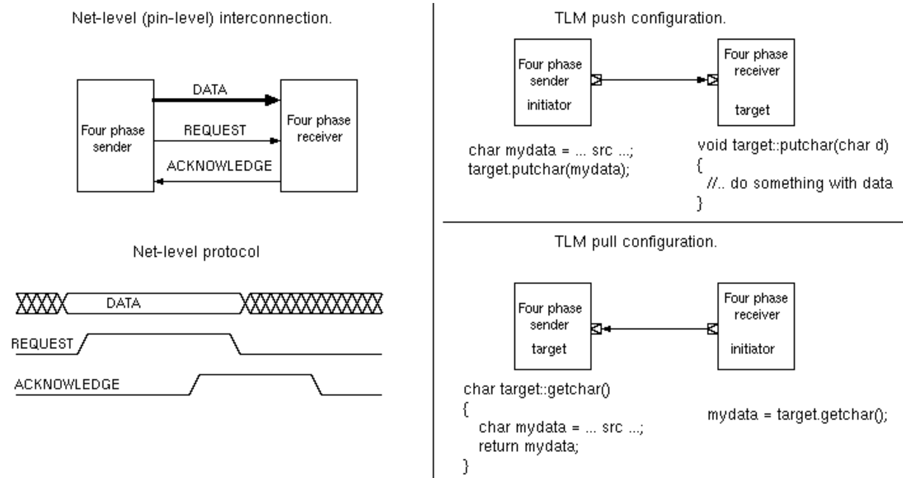


Figure 6.3: Three views of four-phase handshake between sender and receiver: net-level connection and TLM push and TLM pull configurations (untimed).

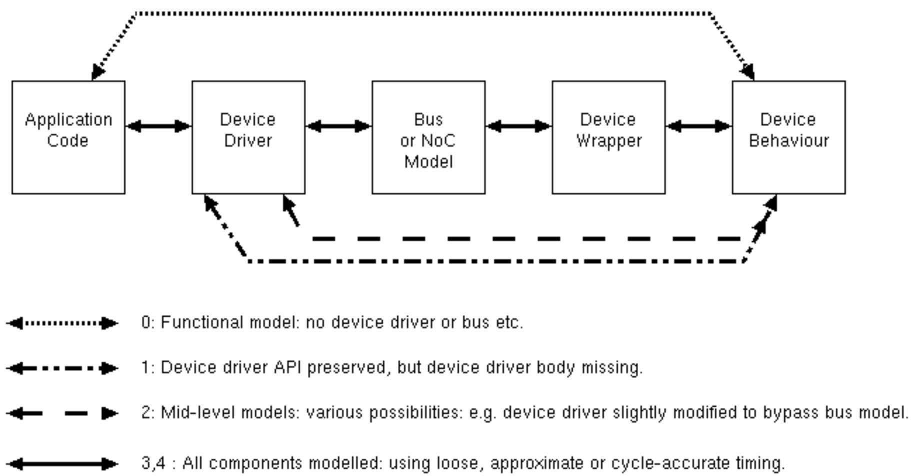


Figure 6.4: Some possible shortcuts through full system model to omit details.

Another view of the higher modelling abstractions:

1. Highest-level (vanished) model: Retains algorithmic accuracy: implemented using SystemC or another threads package: device driver code and device model mostly missing, but the **API to the device driver is preserved**, for instance, a single TLM transaction might send a complete packet when in reality multiple bus cycles are needed to transfer such a packet;

2. Mid-level model: Implemented using SystemC: the device driver is only slightly modified (using preprocessor directives or otherwise) but the interconnection between the device and its driver may be different from reality, meaning bus utilisation figures are unobtainable or incorrect;

3. Bus-transaction accurate mode: each bus operation (read/write or burst read/write and interrupt) is modelled, so bus loading can be established, but timing may be loose and transaction order may be wrong, again, minor changes in the device driver and native compilation may be used;

4. Lower-level models: Implemented in RTL or cycle-accurate SystemC: target device driver firmware and other code is used unmodifed.

Point 3 encompasses mainstream TLM models, like Prazor Virtual PlatformManual

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

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

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

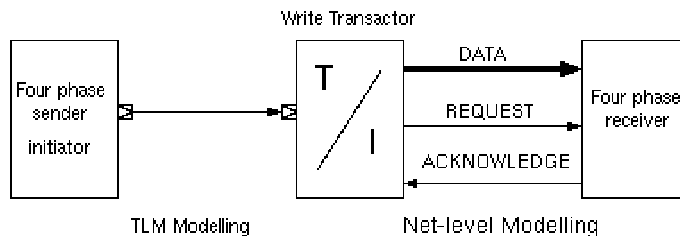


Figure 6.5: Mixing modelling styles using a transactor.

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

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

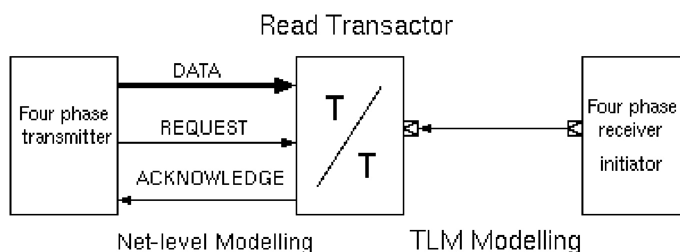


Figure 6.6: Mixing modelling styles using a transactor 2.

6.2.3 Transactor Configurations

This slide/section not lectured in 2017/18.

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

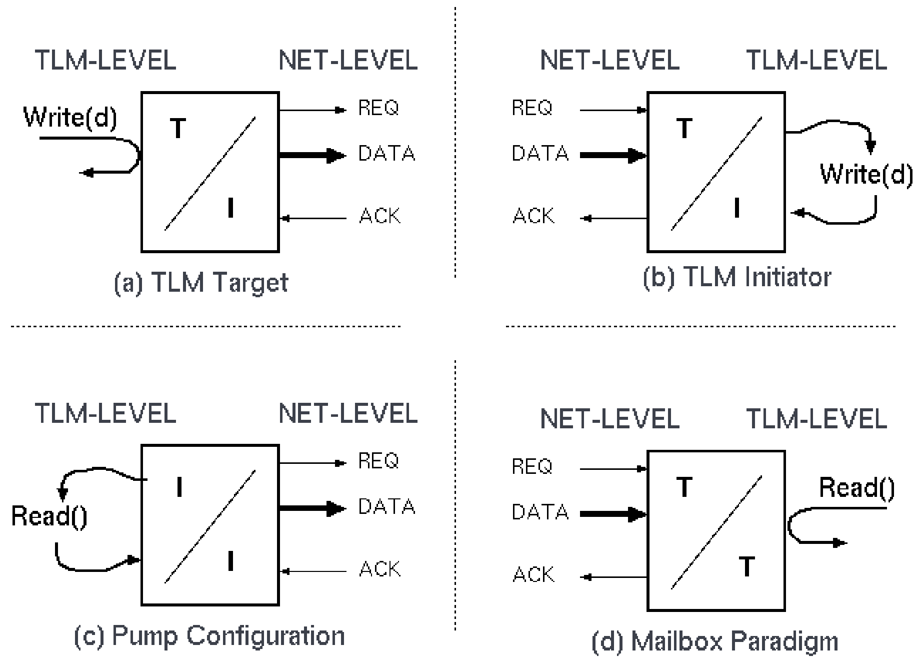


Figure 6.7: Possible configurations for simple transactors.

Additional notes:

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

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

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

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

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

6.2.4 Example of non-blocking coding style:

This slide/section not lectured in 2017/18.

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

CST.

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

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

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

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

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

6.2.5 ESL TLM in SystemC: First Standard TLM 1.0.

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

  // methods to implement ethernet_if
  ...

  // methods to implement usb_if
  ...

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

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

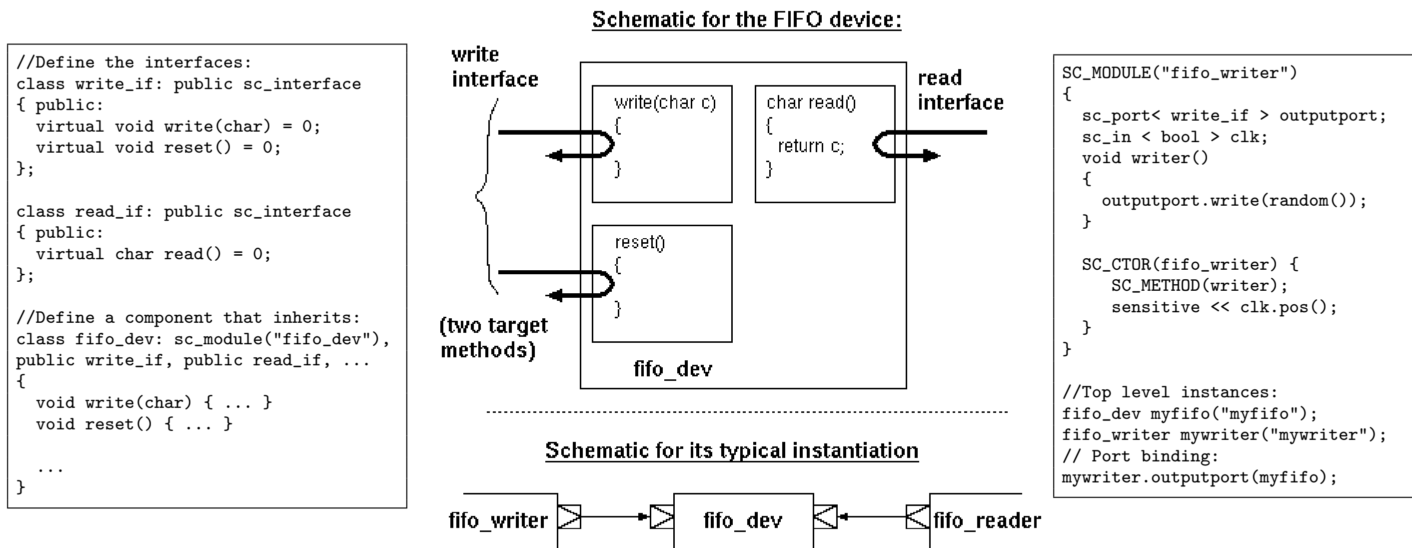
The OSCI TLM 1.0 standard used conventional C++ concepts of multiple inheritance. As shown in the 'Toy ESL' materials and the example here, an SC_MODULE that implements an interface just inherits it.

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

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

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

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



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

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

6.2.6 ESL TLM in SystemC: TLM 2.0

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

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

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

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

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

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

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

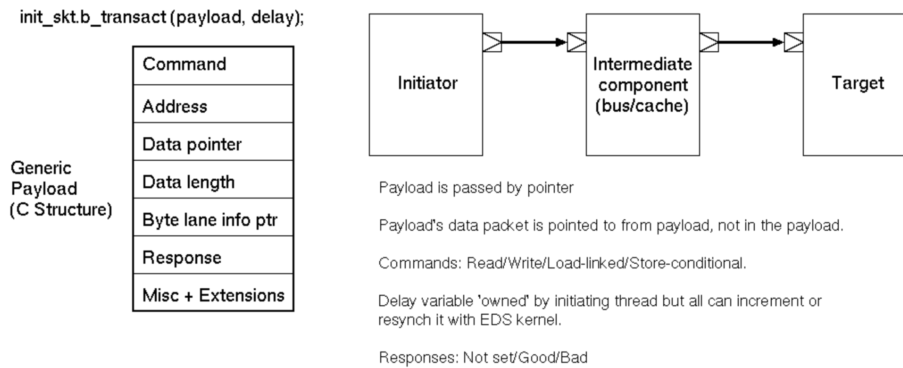


Figure 6.8: The general TLM 2.0 Setup

Here is the constructor:

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

And here is the guts of b_transact:

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

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

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

Wire up the ports in the level above:

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

The full code, and many other examples, can be found in the Prazor virtual platform (see for example

`vhls/src/memories/sram64_cbg.h`).

This slide/section not lectured in 2017/18. (Socket details and types not examinable for Part II CST. The TLM 1.0 style is easier to understand, but not as convenient for real-world projects.)

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

Figure 6.9: Typical setup showing four socket types.

KEY 6. ARCHITECTURAL EXPLORATION USING ESL MODELLING

Additional notes:

TLM 2.0 Socket Types:

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

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

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

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

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

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

6.2.7 Timed Transactions: Adding delays to TLM calls.

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

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

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

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

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

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

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

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

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

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

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

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

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

6.2.8 TLM - Measuring Utilisation and Modelling Contention

This slide/section not lectured in 2017/18.

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

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

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

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

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

6.2.9 Replacing Queues With Delay Estimates

This slide/section not lectured in 2017/18.

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

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

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

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

      last_measure_time = sc_time_stamp();
      opcount = 0;
   }

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

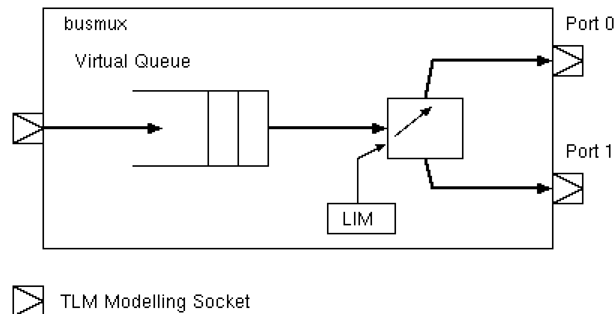


Figure 6.10: Busmux logical schematic diagram.

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

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

6.2.10 Instruction Set Simulator (ISS)

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

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

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

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

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

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

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


      ...
   ...

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

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

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

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

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

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

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

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

6.2.11 Typical ISS setup with Loose Timing (Temporal Decoupling)

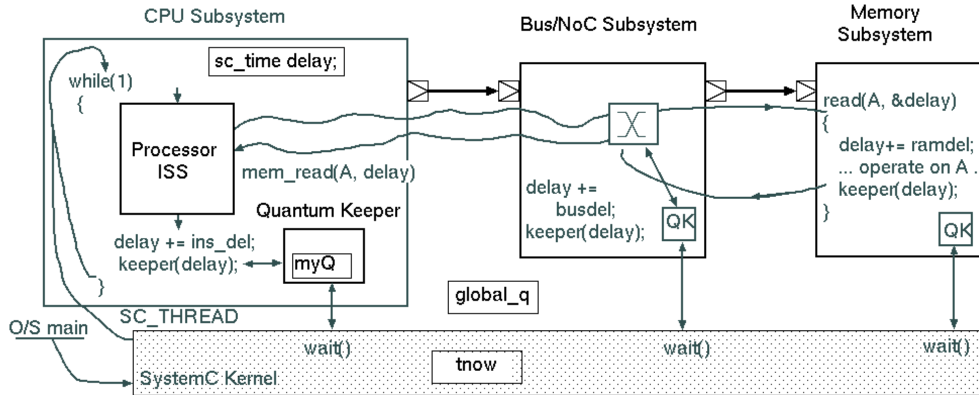


Figure 6.11: Typical setup of thread using loosely-timed modelling with a quantum keeper.

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

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

The quantum keeper code is just a conditional resynch:

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

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

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

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

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

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

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

6.3 Power Estimation: RTL versus ESL

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

6.3.1 RTL Operating Frequency and Power Estimation

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

A table of possible approaches:

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

6.3.2 Gold standard: Power Estimation using Simulation Post Layout

Spreadsheet style power modelling from VCD and SAIF logs.

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

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

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

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

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

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

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

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

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

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

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

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

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

 endmodule
```

But the following dynamic quantities require heuristic estimates:

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

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

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

We here use toggle rates, instead of activity ratios.

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

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

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

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

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

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

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

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

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

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

6.3.4 Typical macroscopic performance equations: SRAM example.

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

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

Technology parameters:

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

Interpolated equations:

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

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

6.3.5 Typical macroscopic performance equations: DRAM example.

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

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

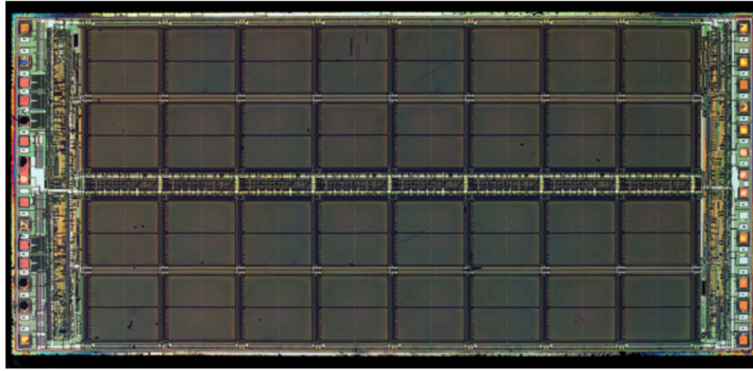


Figure 6.12: Micron Technologys MT4C1024 DRAM chip (1024x1024x1 circa 1994).

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

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

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

Figure 6.13: DRAM activation energies.

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

The Prazor virtual platofrm integrates the University of Maryland DRAM simulator.

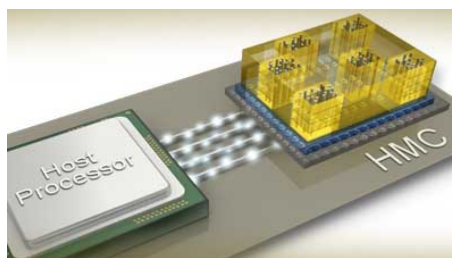


Figure 6.14: Die-stacked DRAM subsystem: The Micron Hybrid Memory Cube.

The phy is the set of pads that operate with high performance to drive the PCB traces. Such power can be minimised if the traces are kept short with using multi-chip modules or die stacking. Micron

have released a multi-channel DRAM module: the Micron HMC. This can have a number of host nodes sharing a number of die-stacked DRAM chips.

There is /will be an exercise on the sheet where DRAM performance and energy need to be computed/estimated.

6.3.6 Rent's Rule Estimate of Wire Length

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

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

Rent's rule uses a simple power-law relationship for the number of external nets to a sub-system as function of the number of logic gates in the sub-system. Figure 6.15 shows three possible design styles that vary in rent coefficient. A circuit composed of components with no local wiring between them is the other extreme possibility, with a Rent exponent of 1.0. But the rule-of-thumb is that for most 'general' subsystems a Rent exponent varying between about 0.5 and 0.7 is seen. Circuits like the shift register can be outliers, having no increase in external connectivity regardless of length: these have a Rent exponent of 0. The same situation arises with an accelerator on-a-stick, where the degree of unfold will alter the rent exponent owing to the fixed-configuration bus port.

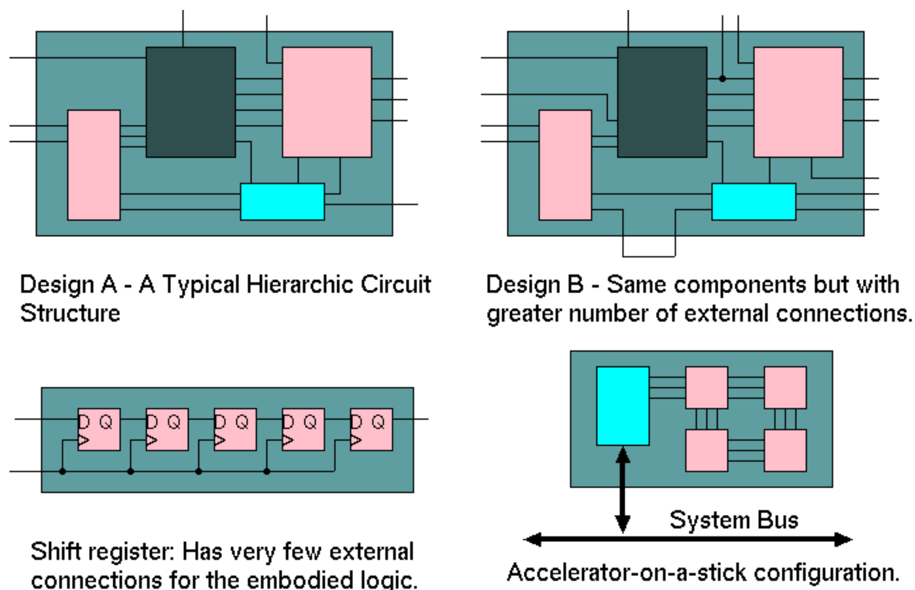


Figure 6.15: Two similar designs with different Rent exponents and two non-Rentian design points.

Lowest-Common Parent Approach Assuming Good Layout:

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

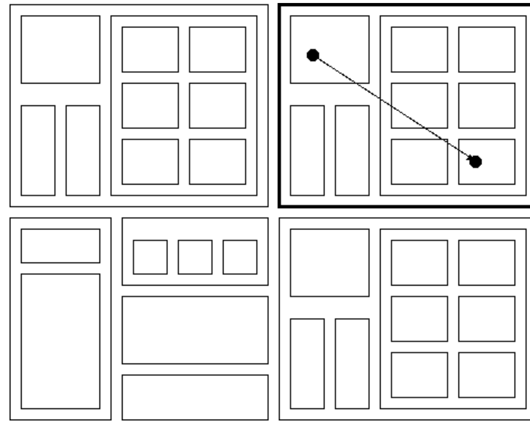


Figure 6.16: Illustrating Lowest Common Parent of the endpoint logic blocks. (This will always be roughly the same size for any sensible layout of a given design, so having a detailed layout like the one shown is not required).

6.3.7 Macroscopic Phase/Mode Power Estimation Formula

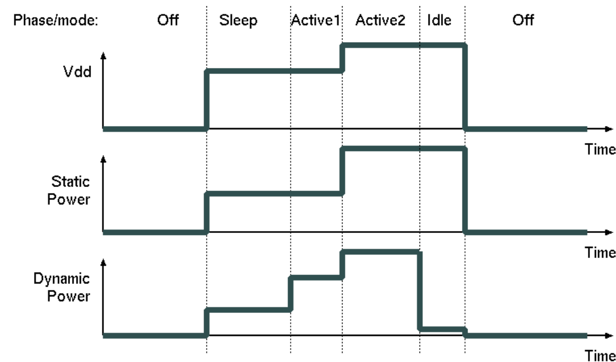


Figure 6.17: Typical Phase and Mode Time-Domain Plot.

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

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

6.3.8 Spreadsheet-based Energy Accounting

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

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

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

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

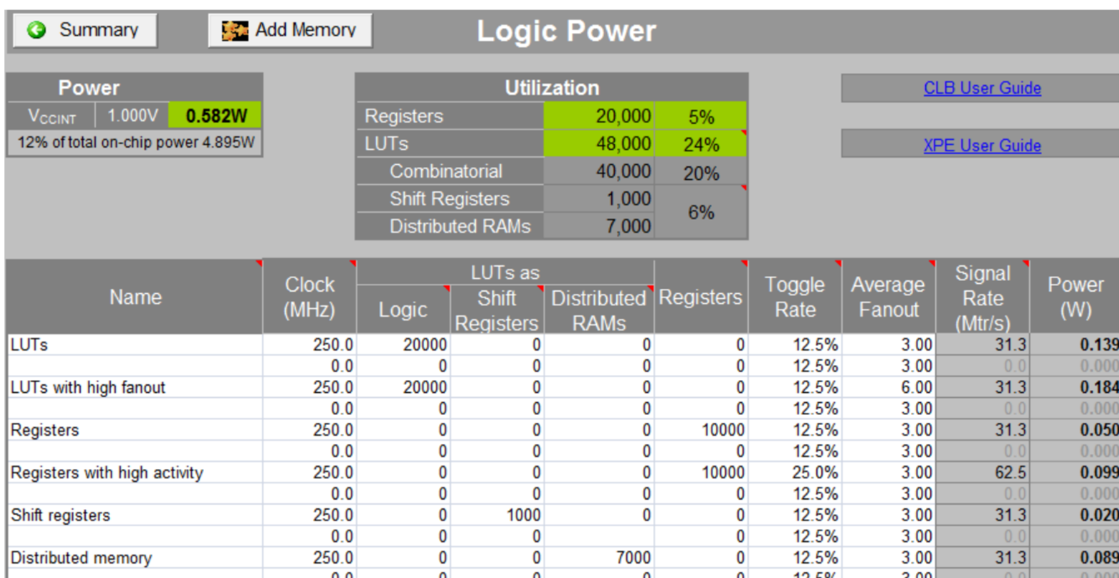
Figure 6.18: Phase/Mode figures for a USB Phy Line Driver.

6.3.9 Higher-Level Simulation - Virtual Platforms

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

End of notes.

©DJ Greaves, Feb 2018.



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

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

Figure 6.19: Example Xilinx Xpower Spreadsheet