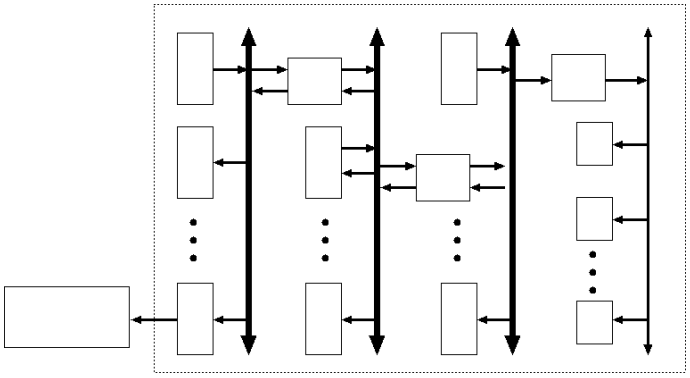
System on Chip Design and Modelling



University of Cambridge
Computer Laboratory
Lecture Notes

Dr. David J Greaves

Part II
Computer Science Tripos
Michaelmas Term 2014/15

- **(1) Basic SoC Components**
- **(2) Power, Performance and Technology**
- **(3) Architectural Design: Partition and Exploration**
- **(4) Verilog RTL: Modules, Protocols and Interfaces**
- **(5) Formal Methods and Assertion-Based Design**
- **(6) SystemC: Hardware Modelling Library**
- **(7) ESL: Electronic System Level Modelling**

0.0.1 SoC Design : 2014/15: Twelve Lectures for CST Part II

A current-day system on a chip (SoC) consists of several different microprocessor subsystems together with memories and I/O interfaces. This course covers SoC design and modelling techniques with emphasis on architectural exploration, assertion-driven design and the concurrent development of hardware and embedded software. 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.)

0.0.2 Recommended Reading

Subscribe for webcasts from 'Design And Reuse': www.design-reuse.com

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.

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

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).

Foster, H.D. & Krolnik, A.C. (2008). *Creating assertion-based IP* . Springer (Series on Integrated Circuits and Systems).

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

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

0.0.3 Example: A Cellphone.

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

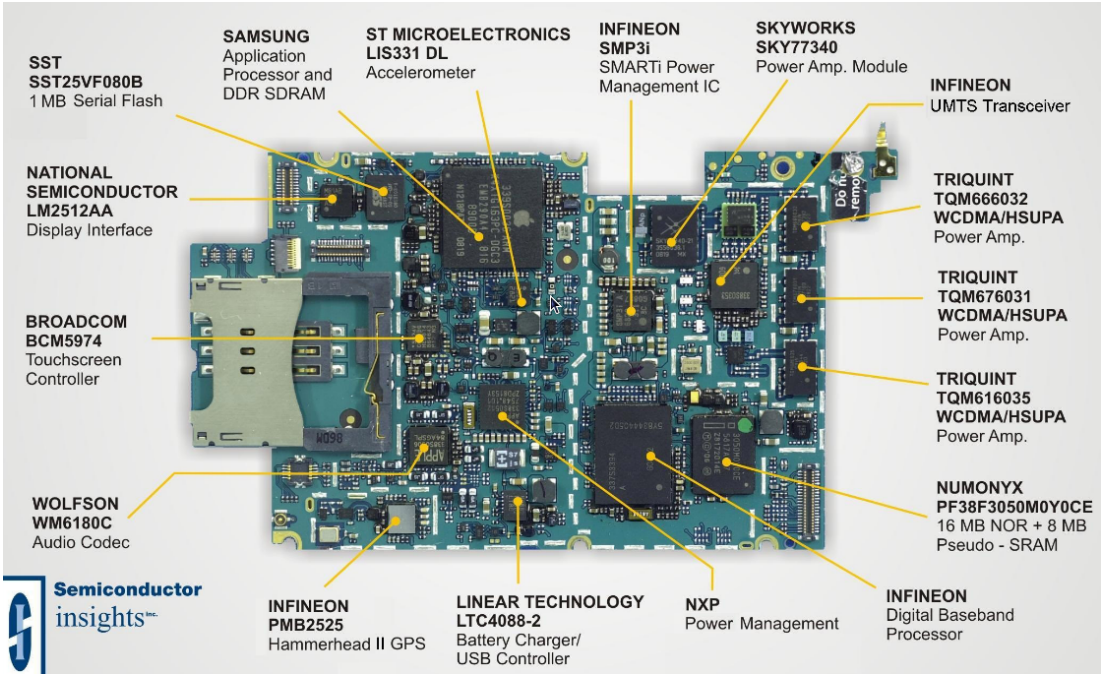


Figure 1: One of the two main PCBs of an Apple iPhone. Main SoC is top, left-centre.

Further examples: iFixit Teardowns iPhone Dissected

Samsung GalaxyCellphone physical components - bill of materials:

- Main SoC - Application Processor, Caches and DRAM
- 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 ?

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

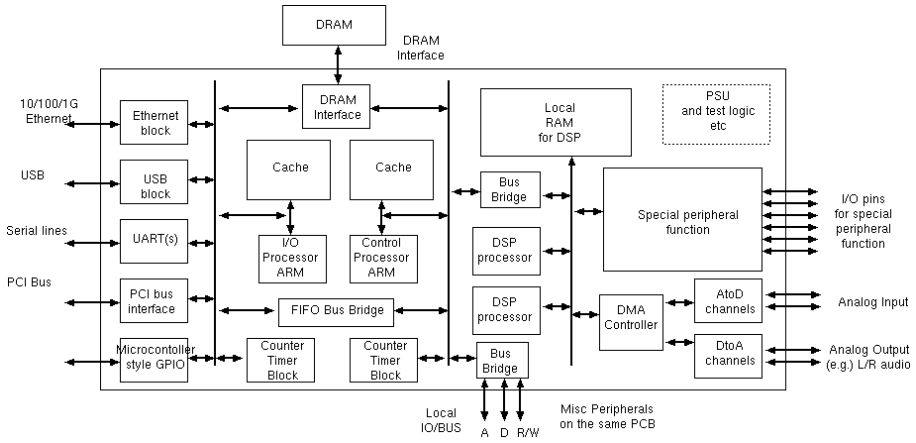


Figure 2: Block diagram of a multi-core 'platform' chip, used in a number of networking products.

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.

0.1 Hardware Design Flow

The hardware design flow is divided by the **Structural RTL** level into:

- **Front End**: specify, explore, design, capture, synthesise $\rightsquigarrow$ **Structural RTL**
- **Back End**: **Structural RTL** $\rightsquigarrow$ 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.

Figure 4 shows a typical design and maufacturing flow that leads from design capture to SoC fabrication.

0.1.1 Front End

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

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.

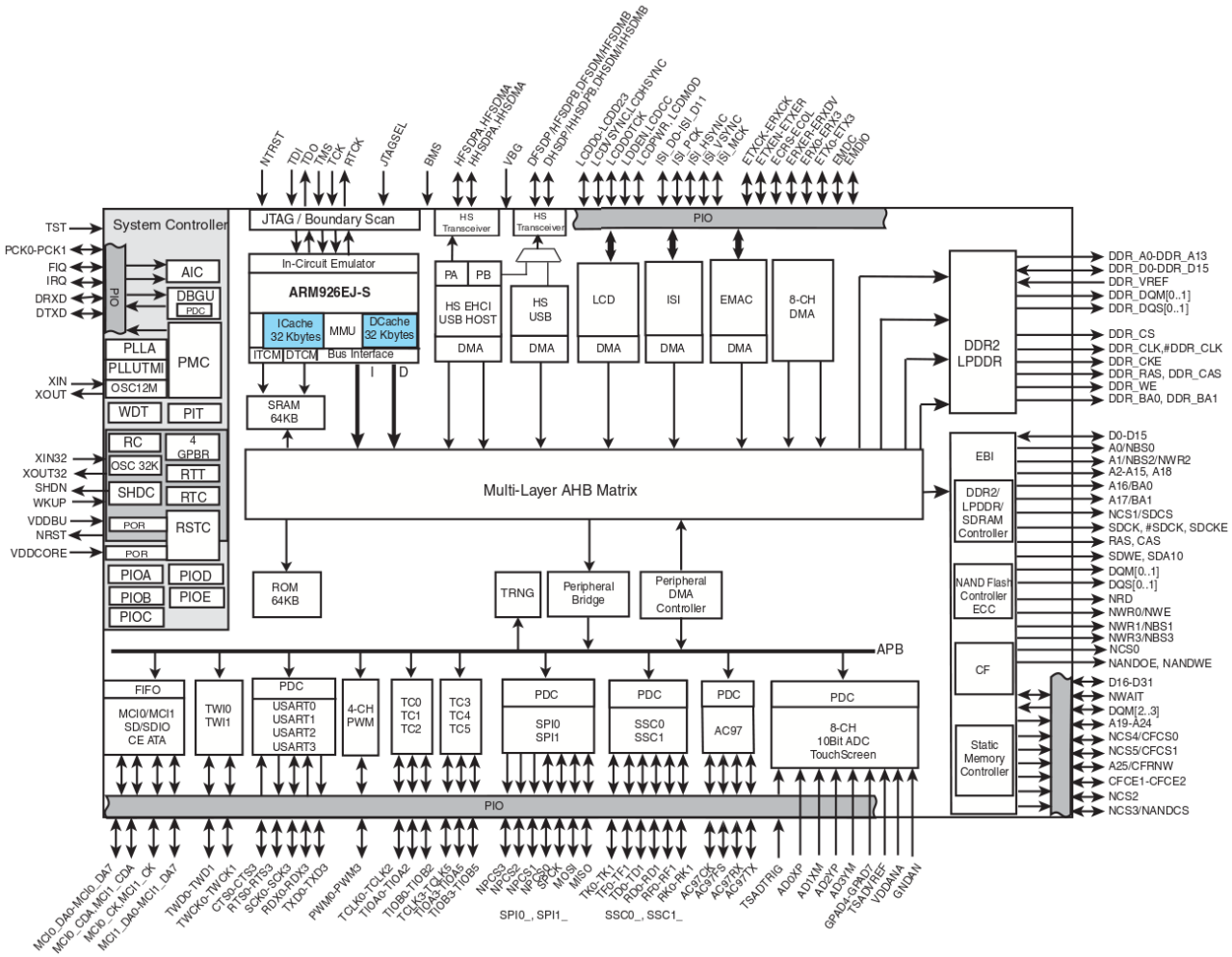


Figure 3: Platform Chip Example: Atmel SAM Series 9645.

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.

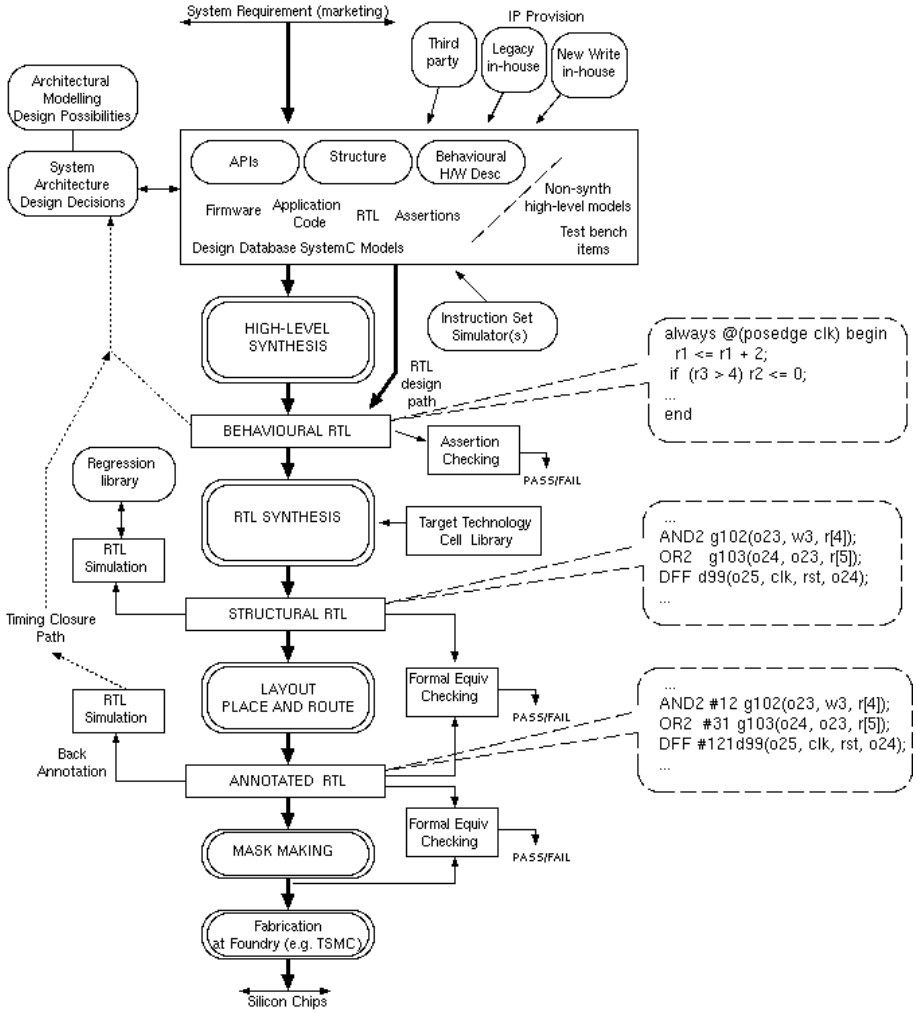


Figure 4: Design and Manufacturing Flow for SoC.

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. We need to mix components using different levels of abstraction in one simulation setup.

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.

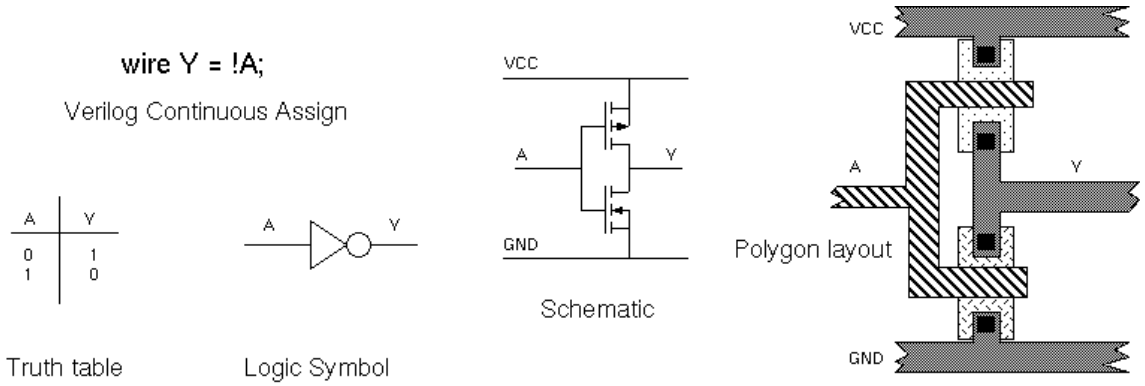


Figure 5: An inverter viewed at various levels of abstraction.

SG 1 — Basic SoC Components

This section is a tour of actual hardware components (IP blocks) found on chips, presented with schematics and illustrative RTL fragments, and connected using a simple bus. Later we will look at other busses and networks on chip.

In the old-fashioned approach, we notice that the hand-crafted RTL used for the hardware implementation has no computerised connection with the firmware, device drivers or non-synthesisable models used for architectural exploration. Today, XML representations of IP-block metainfo resolve this (IP-XACT and OVM/UVM will be mentioned in the last lecture if time permits).

1.1 Simple Microprocessor: Bus Connection and Internals

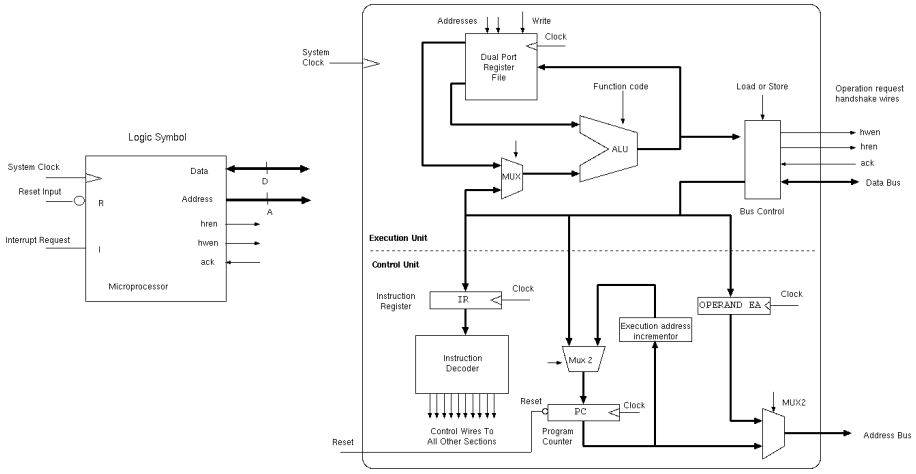


Figure 1.1: Schematic symbol and internal structure for a microprocessor (CPU).

This device is a bus master or *initiator* of bus transactions. It makes a load/read by asserting host read enable: **hren**. It writes to addess space (a store) by asserting host write enable **hwen**. In this course we are concerned with the external connections only.

A central processor unit (CPU) is an execution unit and a control unit. A microprocessor (MPU) is a processor (CPU) on a chip. Early microprocessors such as the original Intel 8080 device had a 16 bit address bus and an 8 bit data bus so can address 64 Kbytes of memory. We say it had an A16/D8 memory architecture. Modern MPUs commonly have on-chip caches and an MMU for virtual memory.

It executes a handshake with external devices using the `hren/hwen` signals as requests and the `ack` signal as an acknowledge. In the following slides every device can respond immediately and so no `ack` signal is shown. In practice, contention, cache misses and operations on slow busses will cause wait states for the processor. Simple processors stall entirely during this period, whereas advanced cores carry on with other work and can receive responses out of order.

The interrupt input makes it save the current PC and load an agreed value that is the entry point for an interrupt service routine.

The high-order address bits are decoded to create chip enable signals for each of the connected peripherals, such as the RAM, ROM and UART.

As we shall see, perhaps the first SoCs, as such, were perhaps the microcontrollers. The Intel 8051 used in the mouse shipped with the first IBM PC is a good example. For the first time, RAM, ROM, Processor and I/O

devices are all on one piece of silicon. We all now have many of these such devices : one in every card in our wallet or purse. Today's SoC are the same, just much more complex.

1.1.1 A canonical D8/A16 Micro-Computer

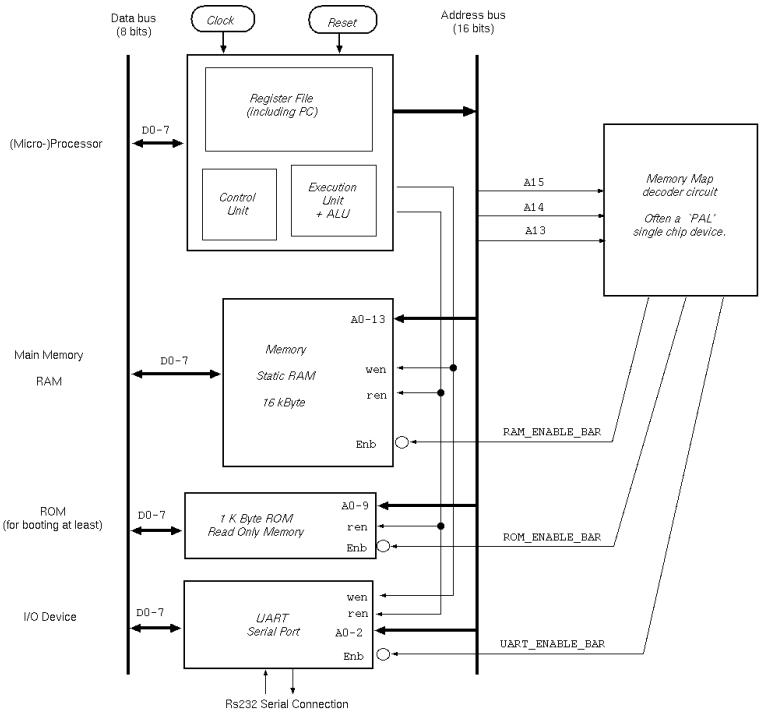
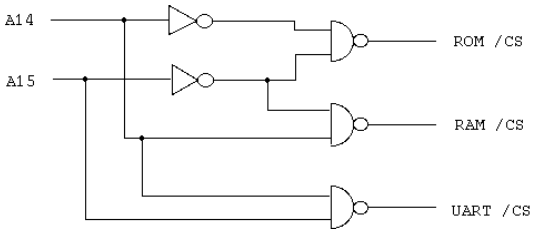


Figure 1.2: Early microcomputer structure, using a bidirectional/tri-state data bus.

Figure 1.2 shows the inter-chip wiring of a basic microcomputer (i.e. a computer based on a microprocessor).



Start	End	Resource
0000	03FF	EPROM
0400	3FFF	Unused images of EPROM
4000	7FFF	RAM
8000	BFFF	Unused
C000	C001	Registers in the UART
C002	FFFF	Unused images of the UART

The following RTL describes the required glue logic for the memory map:

```
module address_decode(abus, rom_cs, ram_cs, uart_cs);
   input [15:14] abus;
   output rom_cs, ram_cs, uart_cs;
   assign rom_cs = (abus == 2'b00);  // 0x0000
   assign ram_cs = (abus == 2'b01);  // 0x4000
   assign uart_cs = !(abus == 2'b11);// 0xC000
endmodule
```

The 64K memory map of the processor has been allocated to the three addressable resources as shown in the table. The memory map must be allocated without overlapping the resources. The ROM needs to be at address zero if this is the place the processor starts executing from when it is reset. The memory map must be known at the time the code for the ROM is compiled. This requires agreement between the hardware and software engineers concerned.

In the early days, the memory map was written on a blackboard where both teams could see it. For a modern SoC, there could be hundreds of items in the memory map. An XML representation called IP-XACT is being adopted by the industry and the glue logic may be generated automatically.

1.1.2 ROM - Read Only Memory

ROM is either mask programmed at manufacture or field-programmable.

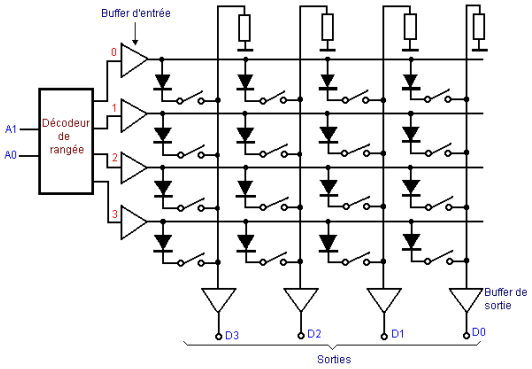


Figure 1.3: ROM Structure - the 'switches' have various implementation technologies.

An addressed row causes the column wires to become one or zero according to whether the diodes are installed (or connected) at the crosspoints. FLASH is a common type of ROM used in USB-sticks and SD cards. The 'switches' in FLASH are transistors with floating gates that are charged and discharged using electron tunnelling when ten or more volts are applied, but which retain their static charge for many years under normal conditions. By 'floating' we mean totally insulated from the rest of the electronic circuit (an envelope of silicon dioxide surrounds each floating gate).

1.1.3 A Basic Micro-Controller

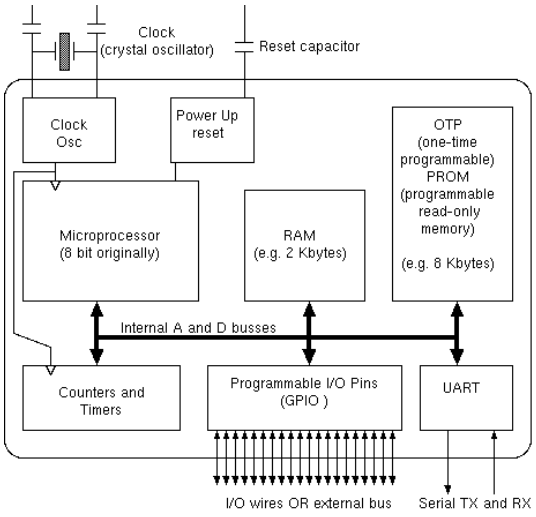


Figure 1.4: A typical single-chip microcomputer (microcontroller).

A microcontroller has all of the system parts on one piece of silicon. First introduced in 1979-85 (e.g. Intel 80C51). Such a microcontroller has an internal D8/A16 architecture and is used in things like a door lock, mouse or smartcard.

Figure 1.5: Contact plate for a smartcard - Reader suppies VCC power, clock and reset. I/O is via the one-bit, bidirectional data pin .

1.1.4 Switch/LED Interfacing

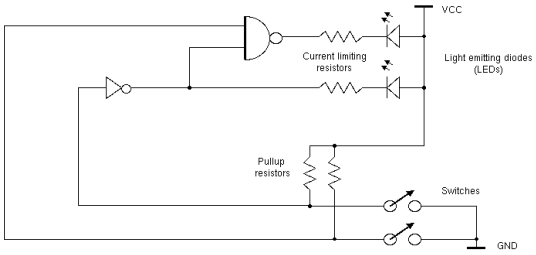


Figure 1.6: Connecting LEDs and switches to digital logic.

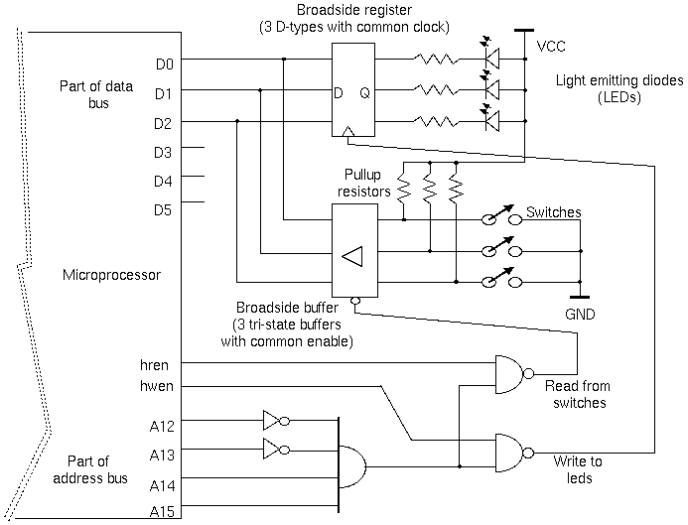


Figure 1.7: Connecting LEDs and switches for CPU programmed I/O (PIO)

Figure 1.6 shows an example wiring structure for hardwired functionality with switches and LEDs. Figure 1.7 shows an example of memory address decode and simple LED and switch interfacing for programmed I/O (PIO) using a microprocessor. When the processor generates a read of the appropriate address, the tri-state buffer places the data from the switches on the data bus. When the processor writes to the appropriate address, the broadside latch captures the data for display on the LEDs until the next write.

1.1.5 UART Device

The RS-232 serial port was widely used in the 20th century for character I/O devices (teletype, printer, dumb terminal). A pair of simplex channels (output and input) make it full duplex. Additional wires are sometimes used for hardware flow control, or a software Xon/Xoff protcol can be used. Baud rate and number of bits per words must be pre-agreed.

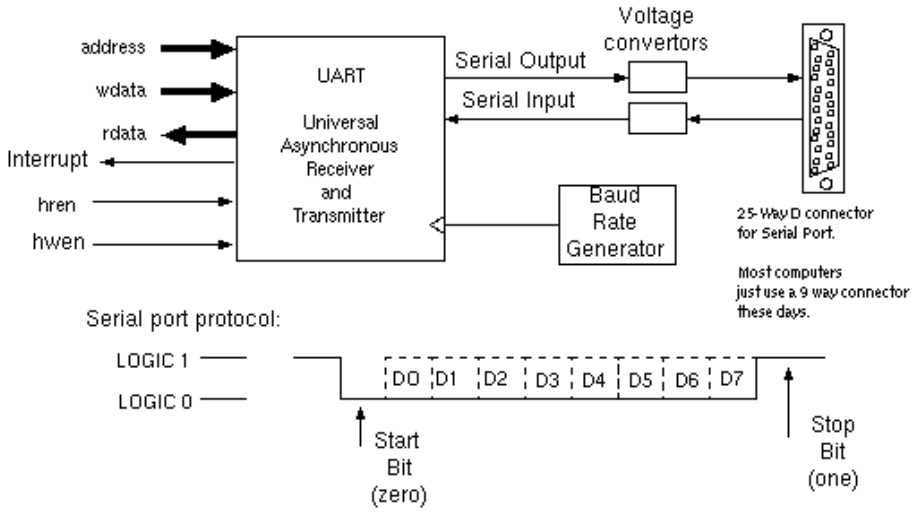


Figure 1.8: Typical Configuration of a Serial Port with UART

1.1.6 Programmed I/O

Programmed Input and Output (PIO). Input and output operations are made by a program running on the processor. The program makes read or write operations to address the device as though it was memory. Disadvantage: Inefficient - too much polling for general use. Interrupt driven I/O is more efficient.

Here is C preprocessor code to define the I/O locations in use by a simple UART device (universal asynchronous receiver/transmitter).

```
//Macro definitions for C preprocessor
//Enable a C program to access a hardware
//UART using PIO or interrupts.

#define IO_BASE 0xFFFC1000 // or whatever


#define U_SEND    0x10
#define U_RECEIVE 0x14
#define U_CONTROL 0x18
#define U_STATUS  0x1C


#define  UART_SEND()  \
   (*((volatile char *)(IO_BASE+U_SEND)))
#define  UART_RECEIVE() \
  (*((volatile char *)(IO_BASE+U_RECEIVE)))
#define  UART_CONTROL() \
  (*((volatile char *)(IO_BASE+U_CONTROL)))
#define  UART_STATUS() \
  (*((volatile char *)(IO_BASE+U_STATUS)))


#define UART_STATUS_RX_EMPTY (0x80)
#define UART_STATUS_TX_EMPTY (0x40)


#define UART_CONTROL_RX_INT_ENABLE (0x20)
#define UART_CONTROL_TX_INT_ENABLE (0x10)
```

The receiver spins until the empty flag in the status register goes away. Reading the data register makes the status register go empty again. The actual hardware device might have a receive FIFO, so instead of going empty, the next character from the FIFO would become available straightaway:

```
char uart_polled_read()
{
   while (UART_STATUS() &
      UART_STATUS_RX_EMPTY) continue;
   return UART_RECEIVE();
}
```

The output function is exactly the same in principle, except it spins while the device is still busy with any data written previously:

```
uart_polled_write(char d)
{
   while (!(UART_STATUS()&
     UART_STATUS_TX_EMPTY)) continue;
   UART_SEND() = d;
}
```

Interrupt driven UART device driver:

```
char rx_buffer[256];
volatile int rx_inptr, rx_outptr;

void uart_reset()
{ rx_inptr = 0;   tx_inptr = 0;
  rx_output = 0;  tx_outptr = 0;
  UART_CONTROL() |= UART_CONTROL_RX_INT_ENABLE;
}
// Here we call wait() instead of 'continue'
// in case the scheduler has something else to run.
char uart_read() // called by application
{ while (rx_inptr==rx_outptr) wait(); // Spin
  char r = buffer[rx_outptr];
  rx_outptr = (rx_outptr + 1)&255;
  return r;
}

char uart_rx_isr()  // interrupt service routine
{ while (1)
    {
      if (UART_STATUS()&UART_STATUS_RX_EMPTY) return;
      rx_buffer[rx_inptr] = UART_RECEIVE();
      rx_inptr = (rx_inptr + 1)&255;
    }
}

uart_write(char c)  // called by application
{ while (tx_inptr==tx_outptr) wait(); // Block if full
  buffer[tx_inptr] = c;
  tx_inptr = (tx_inptr + 1)&255;
  UART_CONTROL() |= UART_CONTROL_TX_INT_ENABLE;
}

char uart_tx_isr()   // interrupt service routine
{ while (tx_inptr != tx_outptr)
    {
      if (!(UART_STATUS()&UART_STATUS_TX_EMPTY)) return;
      UART_SEND() = tx_buffer[tx_outptr];
      tx_outptr = (tx_outptr + 1)&255;
    }
  UART_CONTROL() &= 255-UART_CONTROL_TX_INT_ENABLE;
}
```

This second code fragment illustrates the complete set of five software routines needed to manage a pair of circular buffers for input and output to the UART using interrupts. If the UART has a single interrupt output for both send and receive events, then two of the five routines are combined with a software dispatch between their bodies. Not shown is that the ISR must be prefixed and postfixed with code that saves and restores the processor state (this is normally written in assembler).

1.2 Typical IP Blocks

In this section, we tour a number of IP (intellectual property) blocks. All will be **targets**, most will also generate **interrupts** and some will also be **initiators**. For capacitance reasons, and owing to the small area use of transistors compared with the area used by busses, we normally do not use bi-directional (tri-state) busses within our SoC: instead we use dedicated busses and multiplexor trees. In this section we use the following RTL net names:

- **addr[31:0]**: Input. Selection of internal address - not all 32 bits will be used,
- **hwen**: Input. Asserted during a write from host to target,
- **hren**: Input. Asserted during a read from target to host,
- **wdata[31:0]**: Input. Data to a target when writing/storing,
- **rdata[31:0]**: Output. Data read from target is reading/loading,
- **interrupt**: Output. Asserted by target when wanting attention.

On an **initiator** the net directions will be reversed. For simplicity, in this section, we assume a synchronous bus with no acknowledgement signal, meaning that every addressed target must respond in one clock cycle with no exceptions. Hence a cycle acknowledge handshake signal is not needed.

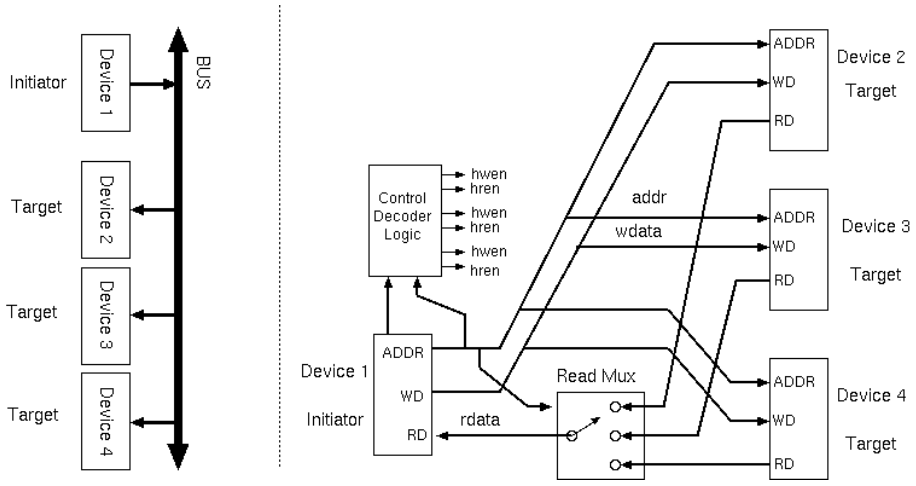


Figure 1.9: A basic SoC bus structure where one initiator addresses three targets (macroview and detailed wiring).

Figure 1.9 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. There is only one initiator, so no bus arbitration is needed.

Max throughput is unity (i.e. one word per clock tick). Typical SoC bus capacity: 32 bits $\times$ 200 MHz = 6.4 Gb/s.

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 just reads or writes of single 32-bit words. In reality, most 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 shared bus standard as such: they can just be dedicated wires running from device to device.

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.

1.2.1 RAM - on chip memory (Static RAM).

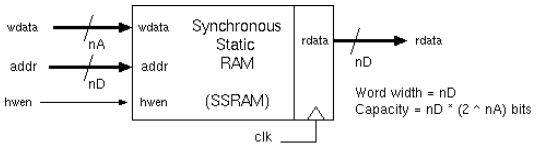


Figure 1.10: Static RAM with single port.

RAMs vary in their size and number of ports. Single-ported SRAM is the most important and most simple resource to connect to our bus. It is a target only. Today's SoC designs have more than fifty percent of their silicon area devoted to SRAM for various purposes.

The 'hren' signal is not shown since the RAM is reading at all times when it is not writing. However, this wastes power, so it would be better to hold the address input stable when not needing to read the RAM.

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. Commonly, synchronous RAMs are used, requiring one clock cycle to read at any address. The same

address can be written with fresh data during the same clock cycle, if desired.

RAMs for SoCs were normally supplied by companies such as Virage and Artizan (but these are now part of larger 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.

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 be made on chip,
- Specialised: 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 Interrupt Wiring: General Structure

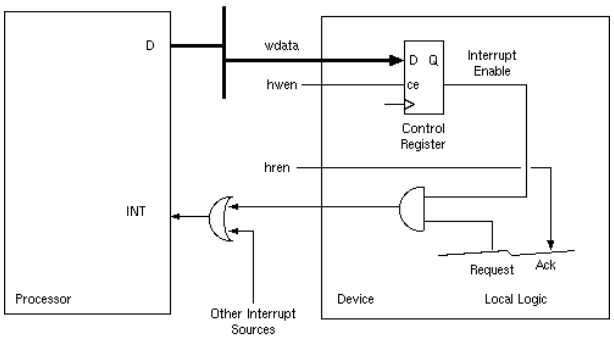


Figure 1.11: Interrupt generation: general structure within a device and at system level.

Nearly all devices have a master interrupt enable control flag that can be set and cleared by under programmed I/O by the controlling processor. Its output is just ANDed with the local interrupt source. We saw its use in the UART device driver, where transmit interrupts are turned off when there is nothing to send.

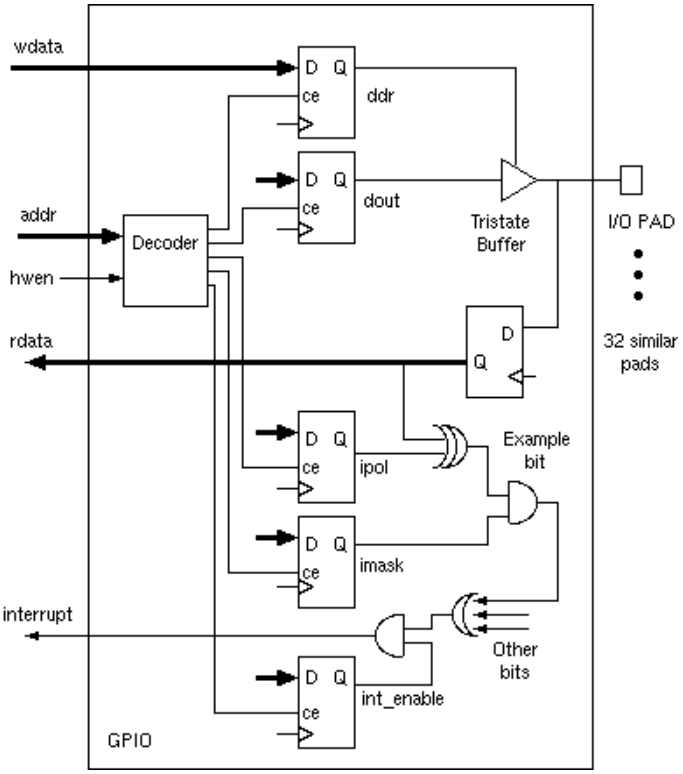
The programmed I/O uses the write enable (`hwen`) signal to guard the transfer of data from the main data bus into the control register. A `hren` signal is used for reading back stored value (shown on later slides).

The principal of programming is (see UART device driver):

- Receiving device: Keep interrupt enabled: device interrupts when data ready.
- Transmit device: Enable interrupt when S/W output queue non-empty: device interrupts when H/W output queue has space.

With only a single interrupt wire to the processor, all interrupt sources share it and the processor must poll around on each interrupt to find the device that needs attention. Enchancement: a **vectored interrupt** makes the processor branch to a device-specific location. Interrupts can also be associated with priorities, so that interrupts of a higher level than currently being run preempt.

1.2.3 GPIO - General Purpose Input/Output Pins



RTL implementation of 32 GPIO pins:

```
// Programming model
reg [31:0] ddr;    // Data direction reg
reg [31:0] dout;   // output register
reg [31:0] imask;  // interrupt mask
reg [31:0] ipol;   // interrupt polarities
reg [31:0] pins_r; // register'd pin data

reg int_enable;// Master int enable (for all bits)


always @(posedge clk) begin
   pins_r <= pins;
   if (hwen && addr==0) ddr <= wdata;
   if (hwen && addr==4) dout <= wdata;
   if (hwen && addr==8) imask <= wdata;
   if (hwen && addr==12) ipol <= wdata;
   if (hwen && addr==16) int_enable <= wdata[0];
   end

// Tri-state buffers.
bufif b0  (pins[0], dout[0], ddr[0]);
.. // thirty others here
bufif b31 (pins[31], dout[31], ddr[31]);

// Generally the programmer can read all the
// programming model registers but here not.
assign rdata = pins_r;

// Interrupt masking
wire int_pending = (|((pins_r ^ ipol)&imask));
assign interrupt = int_pending && int_enable;
```

Microcontrollers have a large number of GPIO pins (see earlier slide).

Exercise: Show how to wire up a push button and sketch out the code for a device driver that returns how many times it has so far been pressed. Sketch polled and interrupt driven code.

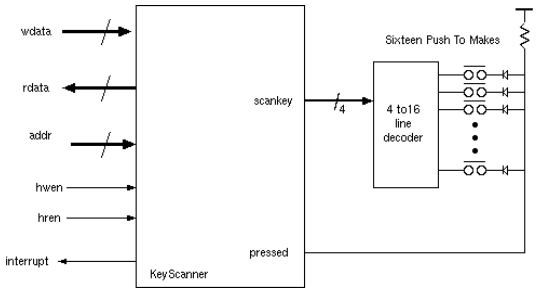
Some state registers inside an I/O block are part of the **programmer's model** in that they can be directly addressed with software (read and/or written), whereas other bits of state are for internal implementation purposes.

The general structure of GPIO pins has not changed since the 6821 controller chip designed in about 1972 that provided 20 such pins. A number of pins are provided that can either be input or output. A data direction register sets the direction on a per-pin basis. If an output, data comes from a data register. Interrupt polarity and masks are available on a per-pin basis for received events. A master interrupt enable mask is also provided.

The slide illustrates the schematic and the Verilog RTL for such a device. All of the registers are accessed by the host using programmed I/O.

1.2.4 A Keyboard Controller

Resistive switches shown (most keyboards and touch screens now use capacitive rather than resistive).




```
output [3:0] scankey;
input pressed;
reg int_enable, pending;
reg [3:0] scankey, pkey;

always @(posedge clk) begin
   if (!pressed) pkey <= scankey;
   else scankey <= scankey + 1;

   if (hwen) int_enable <= wdata[0]
   pressed1 <= pressed;
   if (!pressed1 && pressed) pending <= 1;
   if (hren) pending <= 0;
   end
assign interrupt = pending && int_enable;
assign rdata = { 28'b0, pkey };
```

This simple keyboard scanner scans each key until it finds one pressed. It then loads the scan code into the `pkey` register where the host finds it when it does a programmed I/O read.

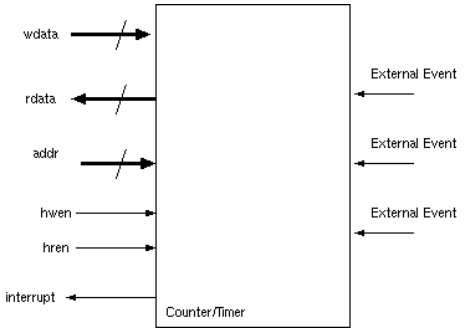
The host will know to do a read when it gets an interrupt. The interrupt occurs when a key is pressed and is cleared when the host does a read `hren`.

The details of this simple sketch are a little unrealistic. In practice, one would not scan at the speed of the processor clock. One would scan more slowly to stop the wires in the keyboard generating radio-frequency interference (RFI). Also, one should use extra register on asynchronous input `pressed` (see crossing clock domains) to avoid metastability. Finally, one would put the keys in a close to square grid, with as many 'pressed' column outputs form the array as row wires feeding the array.

And today, typically, one might use a dedicated microcontroller to scan the keyboard rather than design a hardware circuit.

Note, a standard PC keyboard generates an output byte on press and release and implements a short FIFO internally.

1.2.5 Counter/Timer Block

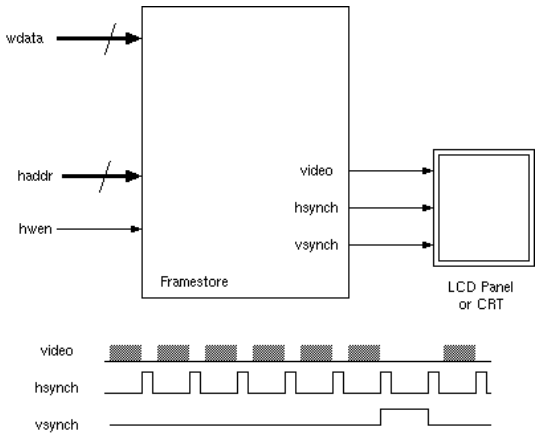



```
// RTL for one channel of a typical timer

//Programmers' Model
reg int_enable, int_pending;

reg [31:0] prescalar;
reg [31:0] reload;

//Internal state
reg ovf'
reg [31:0] counter, prescale;

// Host write operations
always @(posedge clk) begin
   if (hwen && addr==0) int_enable <= wdata[0];
   if (hwen && addr==4) prescalar <= wdata;
   if (hwen && addr==8) reload <= wdata; //FIXED
   // Write to addr==12 to clear interrupt
   end

// Host read operations
assign rdata =
  (addr==0) ? {int_pending, int_enable}:
  (addr==4) ? prescalar:
  (addr==8) ? reload: 0;

// A timer counts system clock cycles.
// A counter would count transitions from external input.
always @(posedge clk) begin
   ovf <= (prescale == prescalar);
   prescale <= (ovf) ? 0: prescale+1;
   if (ovf) counter <= counter -1;
   if (counter == 0) begin
       int_pending <= 1;
       counter <= reload;
       end
   if (host_op) int_pending <= 0;
   end
wire host_op = hwen && addr == 12;

// Interrupt generation
assign interrupt = int_pending && int_enable;
```

The counter/timer block is essentially a counter that counts internal clock pulses or external events and which interrupts the processor on a certain count value. An automatic re-load register accommodates poor interrupt latency, so that the processor does not need to re-load the counter before the next event.

Timer (illustrated in the RTL) : counts pre-scaled system clock, but a counter has external inputs as shown on the schematic (e.g. car rev counter). Four to eight, versatile, configurable counter/timers generally provided in one block. All registers also configured as bus slave read/write resources for programmed I/O. In this example, the interrupt is cleared by host programmed I/O (during `host_op`).

1.2.6 Video Controller: Framestore




```
reg [3:0] framestore[32767:0];
reg [7:0] hptr, vptr;
output reg [3:0] video;
output reg hsynch, vsynch;

always @(posedge clk) begin
     hptr <= (hsynch) ? 0: hptr + 1;
     hsynch <= (hptr >= 230)
     if (hsynch) vptr <= (vsynch) ? 0: vptr + 1;
     vsynch <= (vptr == 110)
     video <= framestore[{vptr[6:0], hptr}];

     if (hwen) framestore[haddr]<= wdata[3:0];
     end
```

The framestore reads out the contents of its frame buffer again and again. The device driver needs to know the mapping of RAM addresses to screen pixels and has zeroed the locations read out during horizontal and vertical synchronisation.

The memory is implemented in a Verilog array and this has two address ports. Another approach is to have a single address port and for the RAM to be simply 'stolen' from the output device when the host makes a write to it. This will cause noticeable display artefacts if writes are at all frequent.

This framestore has fixed resolution and frame rate, but real ones have programmable values read from registers instead of the fixed numbers 230 and 110 (see the linux Modeline tool for example numbers). It is an output only device that never goes busy, so it generates no interrupts.

The framestore in this example has its own local RAM. This reduces RAM bandwidth costs on the main RAM but uses more silicon area. A delicate trade off! A typical compromise, also used on audio and other DSP I/O, is to have a small staging RAM or FIFO in the actual device but to keep as much as possible in the main memory.

Video adaptors in PC computers have their own local RAM or DRAM and also a local processor that performs polygon shading and so on (GPU).

1.2.7 Basic bus: Multiple Initiators.

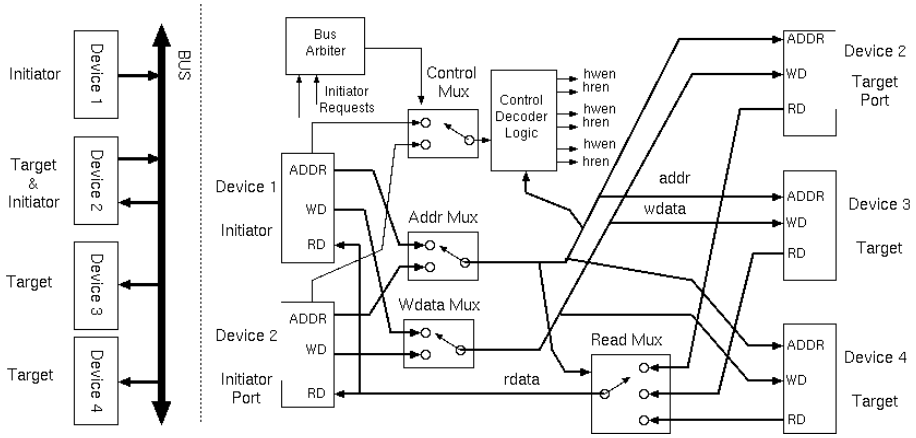


Figure 1.12: SoC bus structure where one of the targets is also an initiator (e.g. a DMA controller).

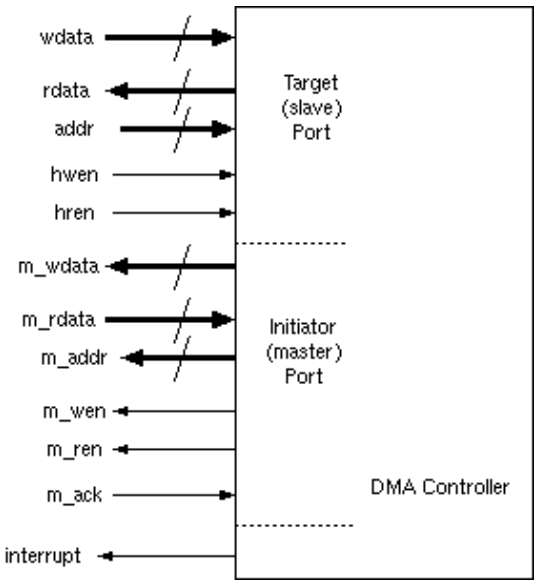
The basic bus may have multiple initiators, so additional multiplexors select the currently active initiator. This

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 others mentioned above.

The maximum bus throughput of unity is now shared among initiators.

Since cycles now take a variable time to complete we 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. Practical busses support bursts of up to, say, 256 words, transferred to/from consecutive addresses. Our simple bus for this section does not support bursts. 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.

1.2.8 DMA Controller



This controller just block copies: may need to keep src and/or dest constant for device access.
DMA controllers may be built into devices: SoC bus master ports needed.

```
// Programmers' Model
reg [31:0] count, src, dest;
reg int_enable, active;

// Other local state
reg [31:0] datareg;
reg intt, rwbar;

always @(posedge clk) begin // Target
   if (hwen && addr==0) begin
       { int_enable, active } <= wdata[1:0];
       intt <= 0; rwbar <= 1;
       end
   if (hwen && addr==4) count <= wdata;
   if (hwen && addr==8) src <= wdata;
   if (hwen && addr==12) dest <= wdata;
   end
 assign rdata = ...// Target readbacks

 always @(posedge clk) begin // Initiator
   if (active && rwbar && m_ack) begin
         datareg <= m_rdata;
         rwbar <= 0;
         src <= src + 4;
         end
   if (active && !rwbar && m_ack) begin
         rwbar <= 1;
         dest <= dest + 4;
         count <= count - 1;
         end
   if (count==1 && active && !rwbar) begin
         active <= 0;
         intt <= 1;
         end
   end
assign m_wdata = datareg;
assign m_ren = active && rwbar;
assign m_wen = active && !rwbar;
assign m_addr = (rwbar) ? src:dest;
assign interrupt = intt && int_enable;
```

The DMA controller is the first device we have seen that is a bus initiator as well as a bus target. It has two complete sets of bus connections. Note the direction reversal of all nets on the initiator port.

This controller just makes block copies from source to destination with the length being set in a third register. Finally, a status/control register controls interrupts and kicks of the procedure.

The RTL code for the controller is relatively straightforward, with much of it being dedicated to providing the target side programmed I/O access to each register.

The active RTL code that embodies the function of the DMA controller is contained in the two blocks qualified with the `active` net in their conjunct.

Typically, DMA controllers are multi-channel, being able to handle four or so concurrent or pending transfers.

Many devices have their own DMA controllers built in, rather than relying on dedicated external controllers. However, this is not possible for devices connected the other side of bus bridges that do not allow mastering (initiating) in the reverse directions. An example of this is an IDE disk drive in a PC.

Rather than using a DMA controller one can just use another processor. If the processor runs out of (i.e. fetches its instructions from) a small, local instruction RAM or cache it will not impact on main memory bus bandwidth with code reads and it might not be much larger in terms of silicon area.

An enhancement might be to keep either of the src or destination registers constant for streaming device access. For instance, to play audio out of a sound card, the destination address could be set to the programmed I/O address of the output register for audio samples and set not to increment.

For streaming media with hard real-time characteristics, such as audio, video and modem devices, a small staging FIFO is likely to be needed in the device itself because the initiator port may experience latency when it is serviced. The DMA controller then initiates the next burst of its transfer when the local FIFO reaches a trigger depth.

1.2.9 Network and Streaming Media Devices

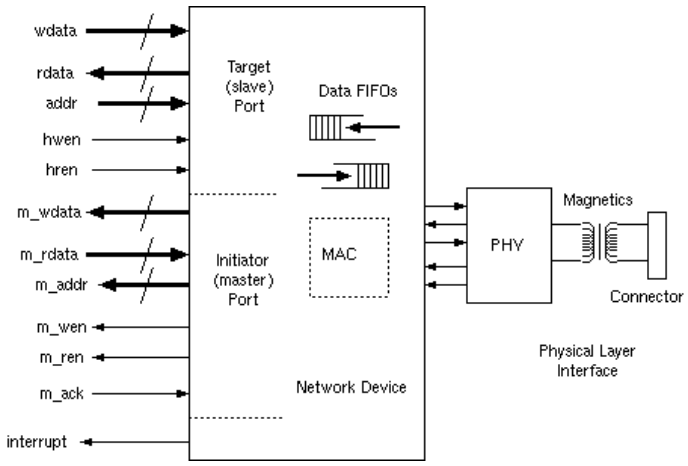


Figure 1.13: Connections to a DMA-capable network device.

Network devices, such as Ethernet, USB, Firewire, 802.11 are similar to streaming media devices, such as audio, and modem devices, and commonly have embedded DMA controllers. Only low throughput devices like the UART are likely not to use DMA.

DMA offloads work from the main processor, but, equally importantly, using DMA requires less staging RAM or data FIFO in device. In the majority of cases, RAM is the dominant cost in terms of SoC area.

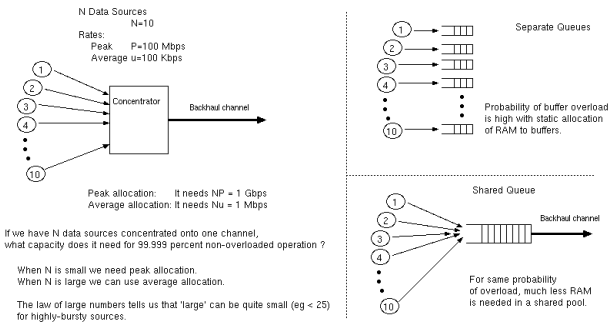


Figure 1.14: Extra diagram illustrating satistical muxltiplexing gain [Non-examinable].

Another advantage of a shared RAM pool is **statistical multiplexing gain**. It is well known in queueing

theory that having a monolithic server performs better than having a number of smaller servers, with same total capacity, that each are dedicated to one client. If the clients all share one server and jobs arrive more or less at random, the system can be more efficient in terms of service delay and overall buffer space needed. The same effect applies to buffer allocation: having a central pool requires less overall RAM, to meet a statistical peak demand, than having the RAM split around the various devices.

The DMA controller in a network or streaming media device will often have the ability to follow elaborate data structures set up by the host CPU, linking and de-linking buffer pointers from a central pool.

1.2.10 Bus Bridge

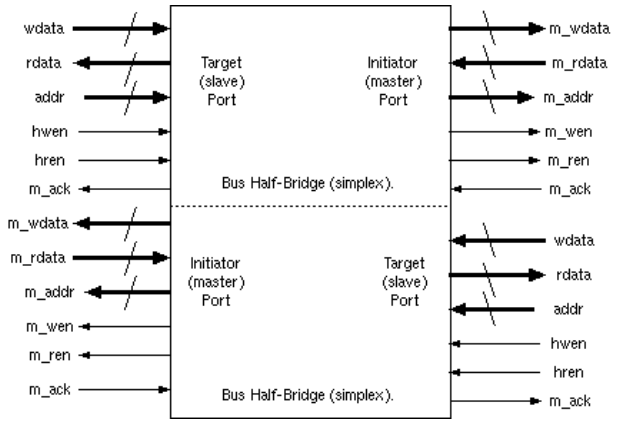


Figure 1.15: Bi-directional bus bridge, composed from a pair of back-to-back simplex bridges.

The essential behaviour of the bus bridge is that bus operations slaved on one side are mastered on the other. The bridge need not be symmetric: speeds and data widths may be different on each side.

A bus bridge connects together two busses that are potentially able to operate independently when traffic is not crossing. However, in some circumstances, especially when bridging down to a slower bus, there may be no initiator on the other side, so that side never actually operates independently and a unidirectional bridge is all that is needed.

The bridge need not support a flat or **unified address space**: addresses seen on one side may be totally re-organised when viewed on the other side or un-addressable. However, for debugging and test purposes, it is generally helpful to maintain a flat address space and to implement paths that are not likely to be used in normal operation.

A bus bridge might implement write posting using an internal FIFO. However it will generally block when reading. In another LG we cover networks on a chip that go further in that respect.

As noted, the 'busses' on each side use multiplexors and not tri-states on a SoC. These multiplexors are different from bus bridges since they do not provide **spatial reuse** of bandwidth. Spatial reuse occurs when different busses are simultaneously active with different transactions.

With a bus bridge, system bandwidth ranges from 1.0 to 2.0 bus bandwidth: inverse proportion to bridge crossing cycles.

1.2.11 Inter-core Interrupter (Doorbell/Mailbox)

The inter-core interrupter (Doorbell/Mailbox) is a commonly-required component for basic synchronisation between separate cores. Used, for instance, where one CPU has placed a message in a shared memory region for another to read. Sometimes the interrupter is part of a central interrupt distributor, such as the 'GIC' from

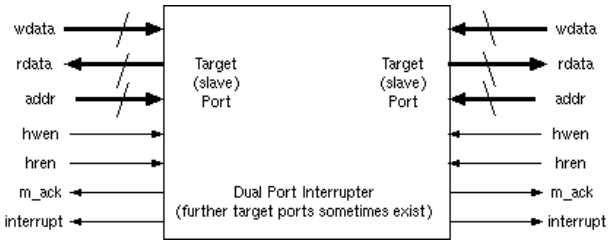


Figure 1.16: Dual-port interrupter (doorbell) or mailbox.

ARM, that enables any device interrupt to be routed to any core with any priority. Such a device offers multiple target interfaces, one per client bus. It generates interrupts to one core at the request of another.

Operational sequence: one core writes a register that asserts an interrupt wire to another core. The interrupted core, in its service routine, reads or writes a register in the interrupter to clear the interrupt.

Mailbox variant allows small data items to be written to a queue in the interrupter. These are read out by the (or any) core that is (or wants to) handle the interrupt. Link: Doorbell Driver Fragments.

1.2.12 Clock Domain Crossing Bridge

A clock-domain-crossing bridge is needed between clock domains. The basic techniques are the same whether implemented as part of an asynchronous FIFO, a SoC bus bridge or inside an IP block (e.g. network receive front end to network core logic). The same techniques apply when receiving asynchronous signals into a clock domain.

The following figure illustrates the key design aspects for crossing in one direction, but generally these details will be wrapped up into something like the domain-crossing FIFO shown elsewhere.

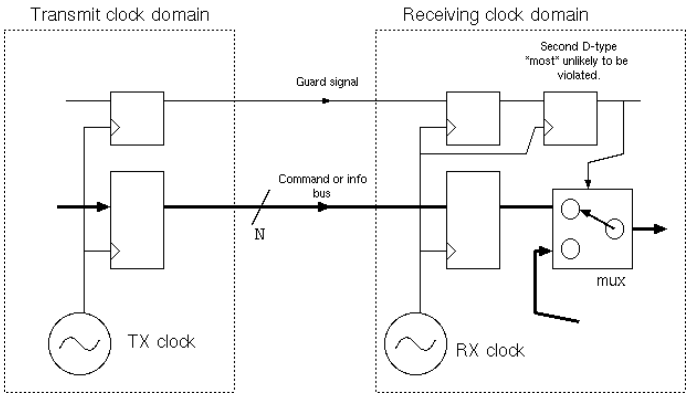


Figure 1.17: Generic setup when sending parallel data between clock domains.

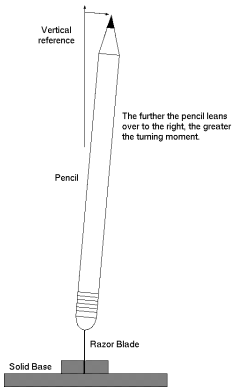
Design principle:

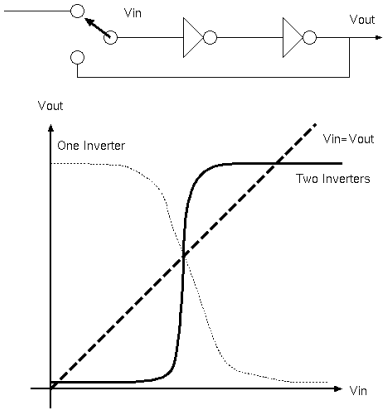
- Have a one-bit signal that is a guard or qualifier signal for all the others going in that direction.
- Make sure all the other signals are settled in advance of guard.
- Pass the guard signal through two registers before using it (metastability avoidance).
- Use a wide bus (crossing operations less frequent).

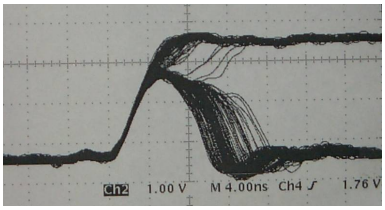
Receiver side RTL:

```
input clk;  // receiving domain clock

input [31..0] data;
input req;
output reg ack;


reg [31:0] captured_data;
reg r1, r2;
always @(posedge clk) begin
     r1 <= req;
     r2 <= r1;
     ack <= r2;
     if (r2 && !ack) captured_data <= data;
```

Metastability Theory:







A pencil balancing on a razor blade can be metastable, but normally flops to one side or the other. A bistable is two inverters connected in a ring. This has two stable states, but there is also a metastable state. If a D-type is clocked while its input is changing, it might be set close to its metastable state and then drift to one level or the other. Sometimes, it will take a fair fraction of a clock period to settle. The oscillogram shows metastable waveforms at the output of a D-type when set/hold times are sometimes violated.

Two quartz crystal oscillators, each of 10 MHz frequency will actually be different by tens of Hz and drift with temperature. Atomic clocks are better: accuracy is one part in ten to the twelve or better, but infeasible to incorporate in everyday equipment and still not good enough to avoid rapid metastable failure.

A simplex clock domain crossing bridge carries information in only one direction. Duplex carries in both directions. Because the saturated symbol rates are not equal on each side, we need a protocol with insertable/deletable padding states or symbols that have no semantic meaning. Or, in higher-level terms, the protocol must have elidable idle states between transactions.

Clock domain crossing is needed when connecting to I/O devices that operate at independent speeds: for example, an Ethernet receiver sub-circuit works at the exact rate of the remote transmitter that is sending to it. Today's microprocessors also have separated clock domains for their cores viz their DRAM interfaces.

The data signals can also suffer from metastability, but the multiplexer ensures that these metastable values never propagate into the main logic of the receiving domain.

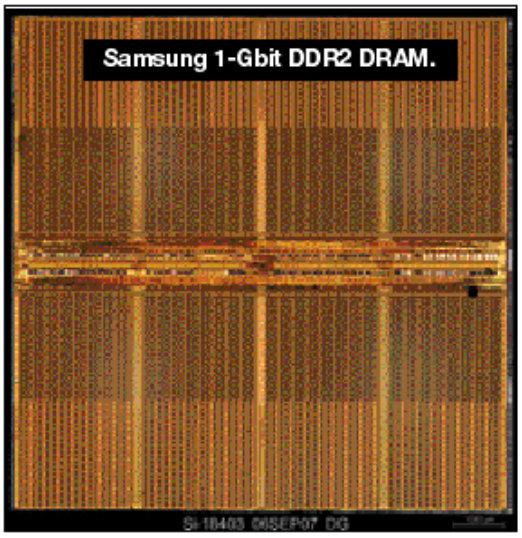
100 percent utilisation is impossible when crossing clock domains. The four-phase handshake limits utilisation to 50 percent (or 25 if registered at both sides) Other protocols can get arbitrarily close to saturating one side or the other provided we know the maximum tolerance in the nominal clock rates. Since clock frequencies are different, 100 percent of one side is either less than 100 percent of the other or else overloaded.

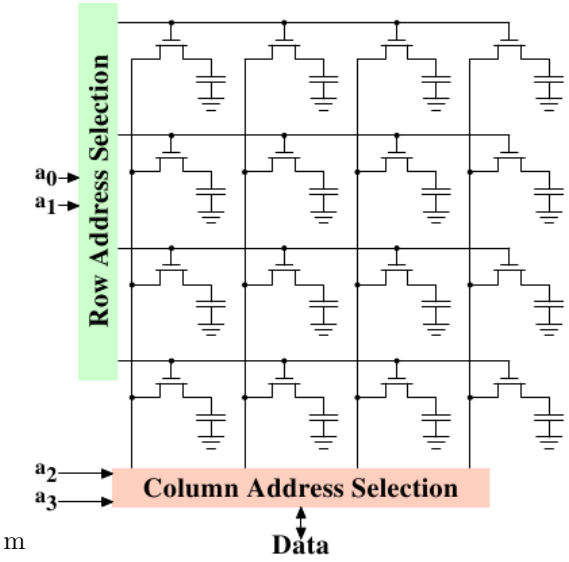
1.2.13 Dynamic RAM : DRAM

Figure 1.18: DRAM single-in-line memory module (SIMM).

DRAMs for use in PCs are mounted on SIMMS or DIMMS, but for embedded applications, often 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 DDR2 instead of DDR3 Also, the most recent SoCs embed some DRAM on the main die or flip-chip bond it right on top of the die in the same package.

Modern DRAM chip with 8 internal memory banks.





Typical DRAM pin connections:

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

High bandwidth: 64 bits times 400 MHz giving 25.6 Gb/s peak. High capacity: Example 1 Gbyte DIMM made of 8 chips. High latency: 20 clock cyles access time to a closed bank. Worse if a bank is already open at the wrong place.

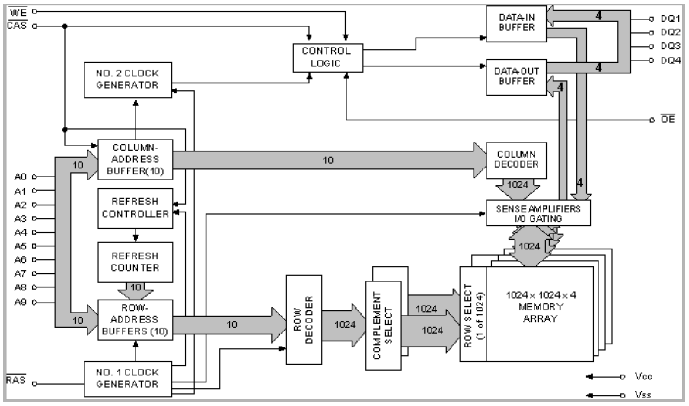


Figure 1.19: Single-bank DRAM Chip Internal Block Diagram.

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 and then 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 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 overhead dictate preference for large burst operations. It is best if clients make available several operations for processing at once: up to number of banks. 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.

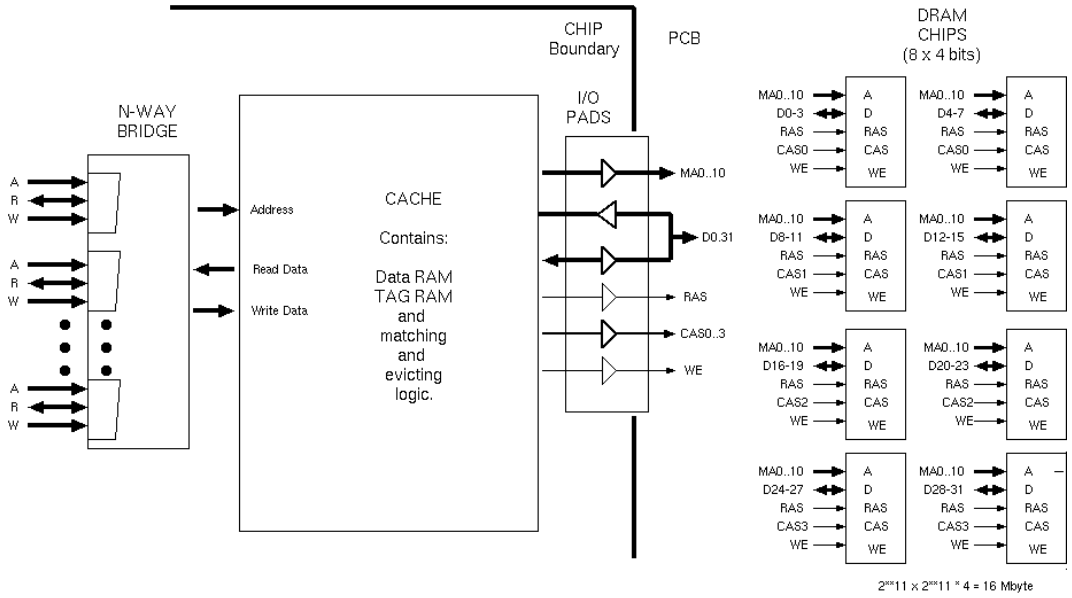


Figure 1.20: Typical structure of a small DRAM subsystem.

Figure 1.20 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 wiring details non-examinable*]

1.2.14 Cache Design

Implementing 4-way, set-associative cache is relatively straightforward. One does not need an associative RAM macrocell: just synthesise four sets of XOR gates from RTL using the '==' operator!

```
reg [31:0] data0 [0:32767], data1 [0:32767], data2 [0:32767], data3 [0:32767];
reg [14:0] tag0 [0:32767], tag1 [0:32767], tag2 [0:32767], tag3 [0:32767];

always @(posedge clk) begin
      miss = 0;
      if (tag0[addr[16:2]]==addr[31:17]) dout <= data0[addr[16:2]];
      else if (tag1[addr[16:2]]==addr[31:17]) dout <= data1[addr[16:2]];
      else if (tag2[addr[16:2]]==addr[31:17]) dout <= data2[addr[16:2]];
      else if (tag3[addr[16:2]]==addr[31:17]) dout <= data3[addr[16:2]];
      else miss = 1;
      end
```

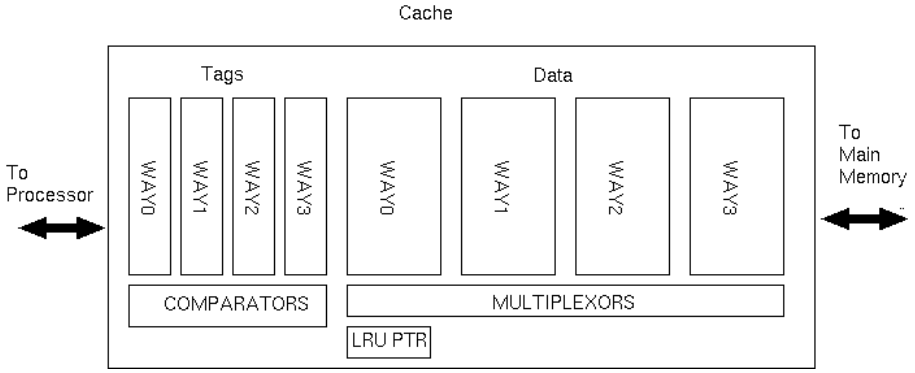


Figure 1.21: Memory blocks and tag comparator needed for a 4-way, set-associative cache.

Of course we also need a write and evict mechanism... (not shown). Rather than implement least-recently-used (LRU) one tends to do 'random' replacement which can be as simple as using keeping a two bit counter to say which 'way' to evict next. Typically an IP company like ARM will provide a high-quality, carefully-tuned implementation, ready to go.

1.2.15 SoC Example: Helium 210

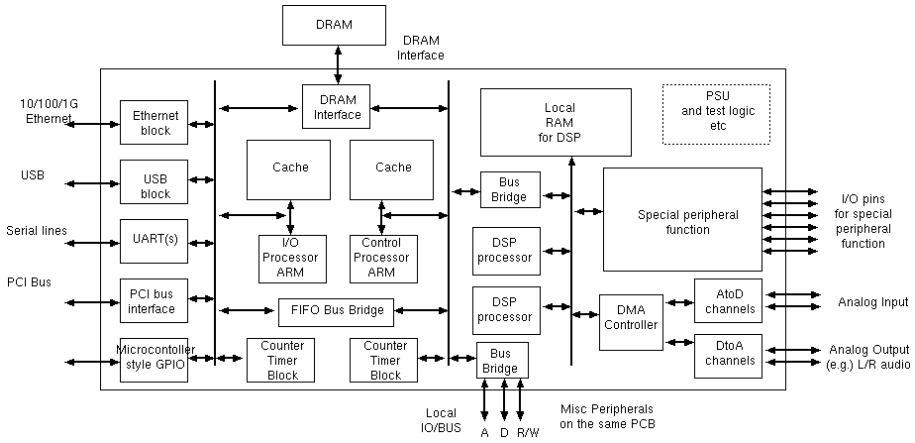


Figure 1.22: Platform Chip Example: Virata Helium 210

A platform chip is the modern equivalent of a microcontroller: it is a flexible chip that be programmed up to serve in a number of embedded applications. The set of components remains the same as for the microcontroller, but each has far more complexity: e.g. 32 bit processor instead of 8. In addition, rather than putting a microcontroller on a PCB as the heart of a system, the whole system is placed on the same piece of silicon as the platform components. This gives us a system on a chip (SoC).

The example illustrated in figure 1.23 has two ARM processors and two DSP processors. Each ARM has a local cache and both store their programs and data in the same off-chip DRAM.

The left-hand-side ARM is used as an I/O processor and so is connected to a variety of standard peripherals. In any typical application, many of the peripherals will be unused and so held in a power down mode.

The right-hand-side ARM is used as the system controller. It can access all of the chip's resources over various bus bridges. It can access off-chip devices, such as an LCD display or keyboard via a general purpose A/D local bus.

The bus bridges map part of one processor's memory map into that of another so that cycles can be executed in the other's space, albeit with some delay and loss of performance. A FIFO bus bridge contains its own

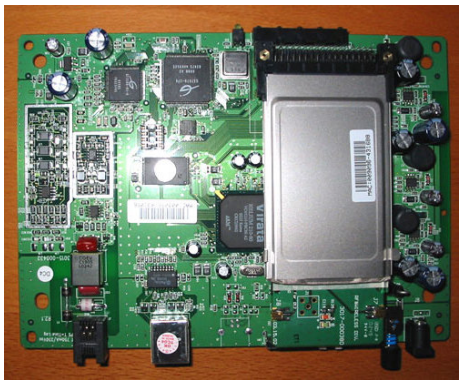


Figure 1.23: Helium chip as part of a home gateway ADSL modem (partially masked by 802.11 module).

transaction queue of read or write operations awaiting completion.

The twin DSP devices run completely out of on-chip SRAM. Such SRAM may dominate the die area of the chip. If both are fetching instructions from the same port of the same RAM, then they had better be executing the same program in lock-step or else have some own local cache to avoid huge loss of performance in bus contention.

The rest of the system is normally swept up onto the same piece of silicon and this is denoted with the 'special function peripheral.' This would be the one part of the design that varies from product to product. The same core set of components would be used for all sorts of different products, from iPODs, digital cameras or ADSL modems.

1.2.16 SoC Example: Atmel SAM9645

The SAM9645 integrates a 400 MHz ARM core and a large number of DMA-capable peripheral controllers using a central bus 'matrix': PDF DataSheet

A platform chip is an SoC that is used in a number of products although chunks of it might be turned off in any one application: for example, the USB port might not be made available on a portable media player despite being on the core chip.

At the architectural design stage, to save the cost of a full crossbar matrix interconnect, devices can be allocated to busses with knowledge of the expected access and traffic patterns. Commonly there is one main bus master per bus. The bus master is the device that generates the address for the next data movement (read or write operation).

Busses are connected to bridges, but crossing a bridge has latency and also uses up bandwidth on both busses. So we should allocate devices to busses so that inter-bus traffic is minimised based on a priori knowledge of likely access patterns.

Lower-speed busses may go off chip.

DRAM is always an important component that is generally off chip as a dedicated part. Today, some on-chip DRAM is being used in SoCs.

1.3 Architecture: Bus and Device Structure

In this section we examine the basic anatomy of a SoC.

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

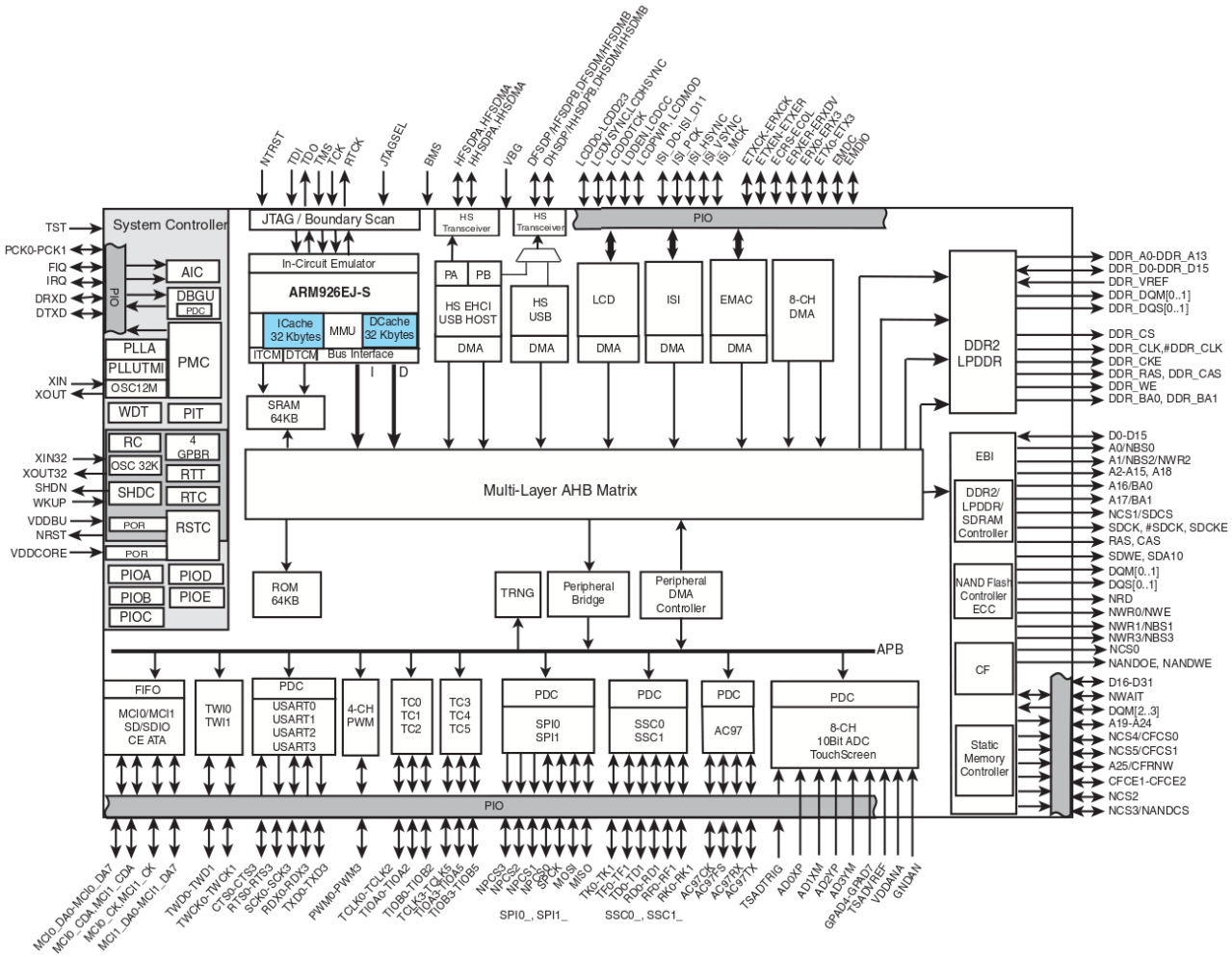


Figure 1.24: Platform Chip Example: Atmel SAM Series 9645.

- 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 requires 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.
- DRAM is several centimeters 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.

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

1.3.1 Basic Bus: One initiator (II).

The bus protocol in the earlier slides that used **addr, hwen, hren, wdata** and **rdata** does not tolerate registering for reads, but if a **ready** or other acknowledgement signal were added, it would be like the four phase handshake and work correctly, but poorly for long distances over the chip.

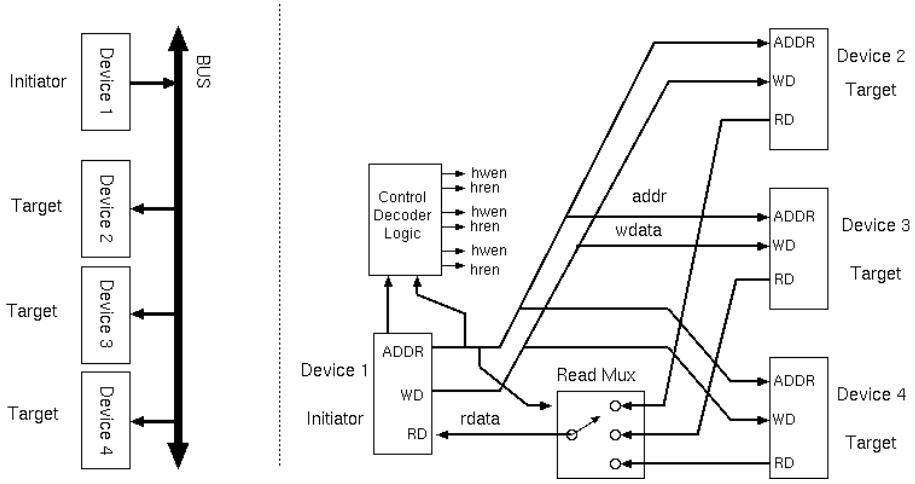


Figure 1.25: Example where one initiator addresses three targets.

Figure 1.25 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.

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.

1.3.2 Basic bus: Multiple Initiators (II).

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).

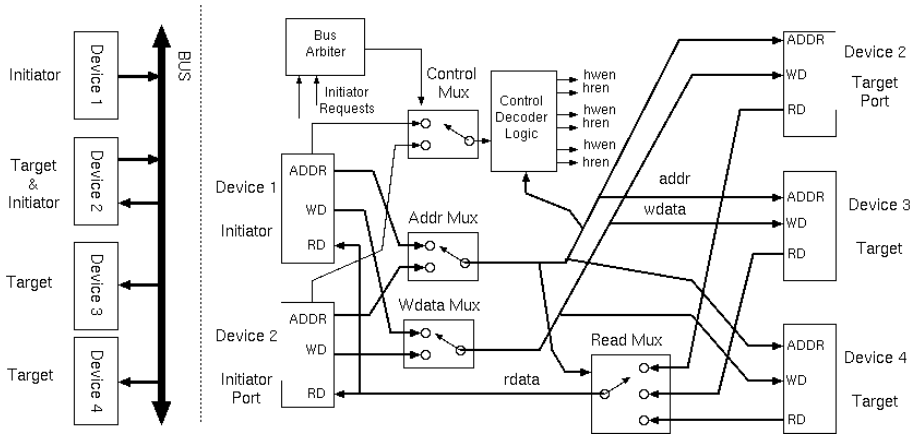


Figure 1.26: Example where one of the targets is also an initiator (e.g. a DMA controller).

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.

1.3.3 Bridged Bus Structures.

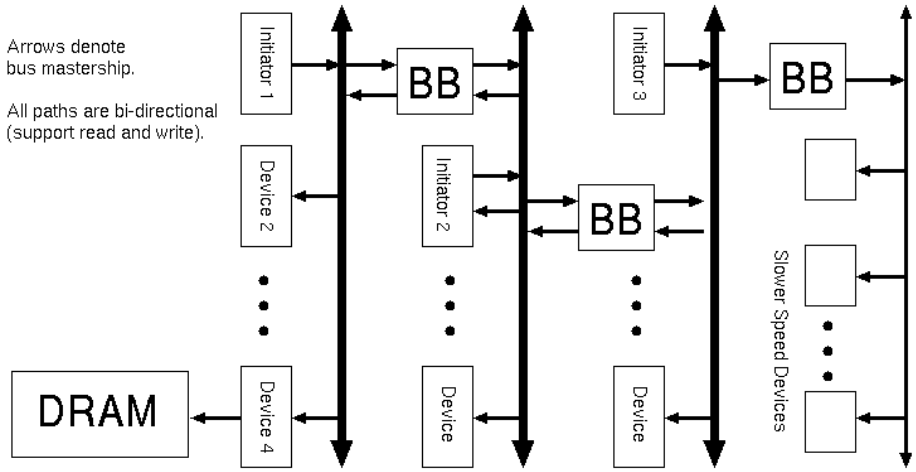


Figure 1.27: A system design using three main busses.

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 and top-levels of structural wiring automatically generated. An example tool that does this is ARChitect2 from ARC International (now part of Virage Logic).

1.3.4 Classes of On-Chip Protocol

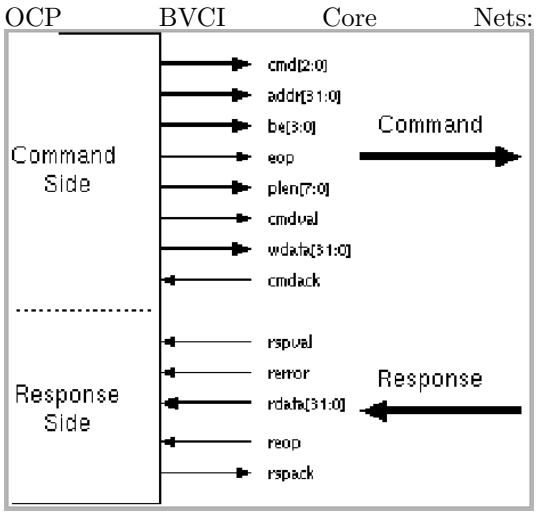
1. Reciprocally-degrading: such as handshake protocols studied earlier: throughput is inversely proprotional to target latency in terms of clock cycles,
2. Delay-tolerant: such as AMBA-3 (ARM's AXI) and OCP's BVCI (below): new commands may be issued while awaiting responses from earlier,
3. Reorder-tolerant: responses can be returned in a different order from command issue: helpful for DRAM access and needed for advanced NoC architectures.
4. Virtual-circuit flow controlled: (beyond scope of this course): each source has a credit counter controlling how many packets it can send and priority mechanisms ensure responses are returned without deadlock.

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.

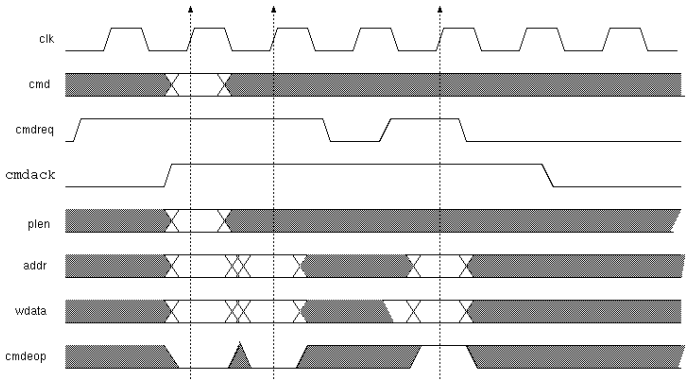
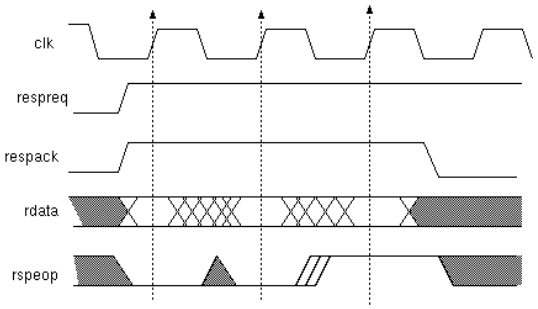


Figure 1.28: BVCI Protocol, Command Timing Diagram

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

1.3.5 Network on Chip: Simple Ring.

A two-level heirarchy of bridged rings is sometimes a sweetspot for SoC design. For example, IBM Cell Broadband Engine uses dual rings. At moderate size, using a fat ring (wide bus links) is better than a thin X-bar design for same throughput in terms of power consumption and area use.

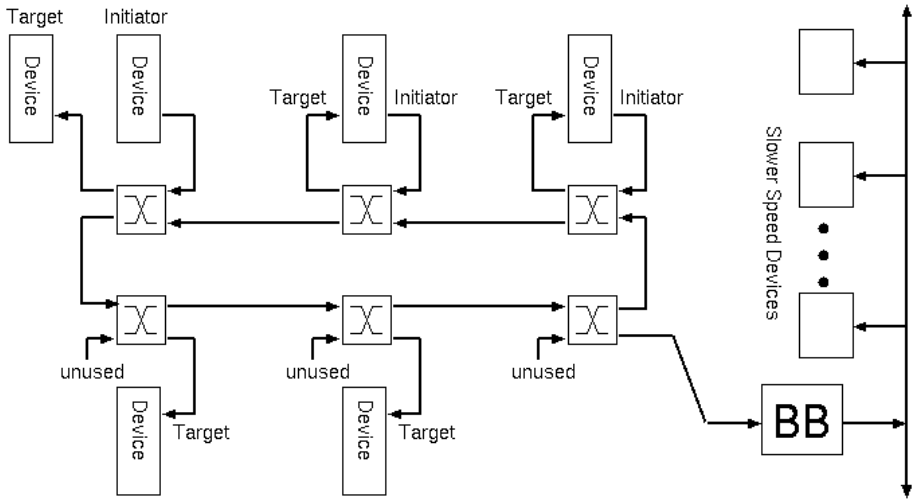


Figure 1.29: A ring network: a low-complexity network on chip structure.

A two-by-two switch element enables formation of rings (and other NoC structures). The switch element is registered: hence ring network can span the chip. A higher-radix element allows more devices to be connected at a 'station'. Performance: Single ring: throughput=2. Dual counter-rotating rings: throughput=4.

With ring (and certainly with all more complex NoCs) IP block protocol/interface needs to support decoupled requests and response packets.

Ring has local arbitration in each element, but global policies are required to avoid deadlock and starvation.

Ring gives priority to traffic already on the ring and uses LAN-like buffering at source, hence no requirement for queuing in element.

Ring does not carry interrupts or other sideband signals.

Switched networks require switching elements. With a 2x2 element it is easy to build a ring network. The switching element may contain buffering or it may rely on back-pressure to make sources reduce their load.

Single ring: throughput=2. Counter-rotating ring (one ring in each direction): throughput=4 since a packet only travels 1/4 of the way round the ring on average.

Using a network, the delay may be multiple clock cycles and so a **write posting** approach is reasonable. If an initiator is to have multiple outstanding read requests pending it must put a token in each request that is returned in the response packet for identification purposes.

Although there can be effective local arbitration in each element, a network on a chip can suffer from deadlock. Some implementations uses separate request and response networks, so that a response is never held up by new requests, but this just pushes deadlock to the next higher logical level when some requests might not be servicable without the server issuing a subsidiary request to a third node. Global policies and careful design are required to avoid deadlock and starvation.

1.3.6 Network on chip: Switch Fabrics.

A simple ring is not very effective for above small tens of nodes. Instead, richer meshes of elements are used and the elements can have a higher radix, such as 4x4.

There are a number of well-known switch wiring schemes, whth names such as Benes, Clos, Shuffle, Delta, Torus, Mesh, Express-Mesh, Butterfly. These vary in terms of the complexity and contention ratios. Note even a full-crossbar (any input to any output in unit time), which is very costly, still suffers from output port contention, so rarely justified on performance grounds, but uniform access delays make it easy to provide sequential consistency (see my Comparative Architecture notes).

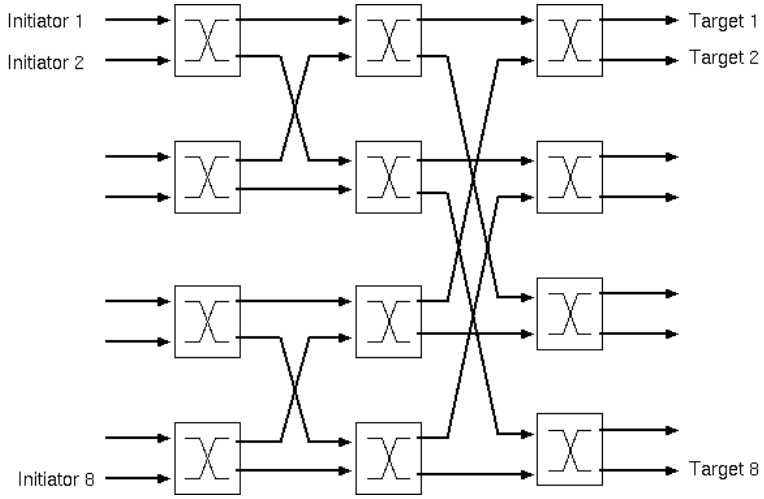


Figure 1.30: A more-complex switching fabric: more wiring, more bandwidth and less fabric contention than ring (but still has output port contention).

Illustrated is using two-by-two switch element connects eight devices in three stages. Using a higher-radix (e.g. 4) is common. The throughput is potentially equal to the number of ports, but the fabric may partially block

and there may be uneven traffic flows leading to receiver contention. These effects reduce throughput. Typically will not need quite as many initiators as targets, so a symmetric switch system will be over provisioned.

Can be overly complex on the small scale, but scale ups well. See Network On Chip Synthesis Tool: Mullins NetGen Network Generator. RDM NoC Notes

1.3.7 Network on Chip: Higher Dimensions.

(*Not examinable for part II*).

Can we consider higher-dimensional interconnect ? The hypercube has lowest diameter for number of customers. But it has excessive wiring. Chips are two-dimensional so perhaps it's good to use a 2-D network ? But this may be overly conservative. Maybe use 2.5-D ? have a small number of 'multi-hop' links?

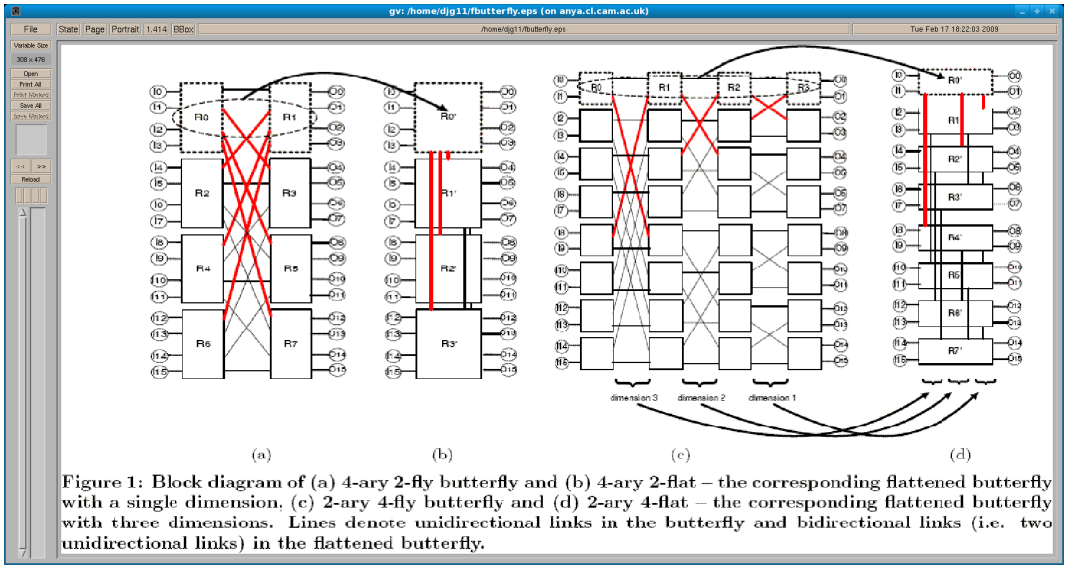


Figure 1.31: The 'Flattened Butterfly' network topology.

On benign (load-balanced) traffic, the flattened butterfly approaches the cost/performance of a butterfly network and has roughly half the cost of a comparable performance clos network.

Further details (non examinable): "The advantage over the clos is achieved by eliminating redundant hops when they are not needed for load balance." See '*Flattened butterfly : a cost-efficient topology for high-radix networks*' by John Kim, William J. Dally, Dennis Abts.

The ARM AXI bus is widely used and can be used with non-ARM products ARM AXI. One ARM AXI protocol includes tags on each operation for out-of-order request/response association: hence it is suitable for pipelined, on-chip networks where message sequencing may vary.

Other busses: The Wishbone bus and IBM CoreConnect bus: used by various public domain IP bocks and various designs (e.g. RTL OpenRISC). The OpenRISC in the practical materials on the course web site uses Wishbone. Wikipedia Wishbone Core Connect

SG 2 — Power, Performance and Technology

Battery life is very important for convenience. Saving energy in computing is always a good idea. In this SG we will examine energy and performance and energy saving techniques.

2.0.8 Basic Physics

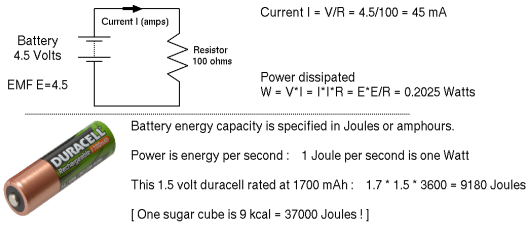


Figure 2.1: Ohms Law, Power Law and Battery Capacity.

2.0.9 Chip Dissipation

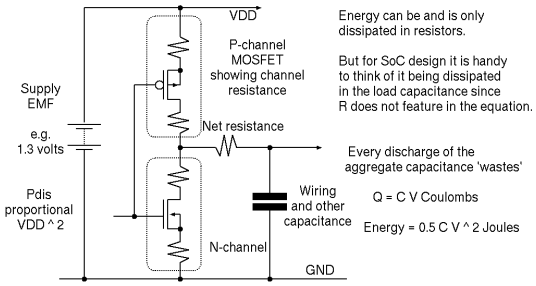
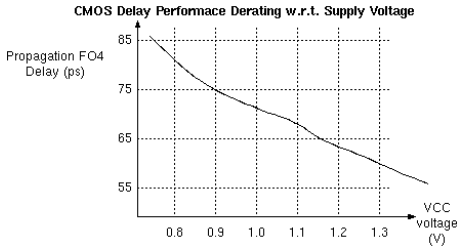


Figure 2.2: Dynamic energy dissipation mechanism.

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 logic one to zero transistion.

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

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



The FO4 delay is the delay through an inverter that is feeding four other nearby inverters (fan out of four).

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

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

2.0.10 Detailed Delay Model.

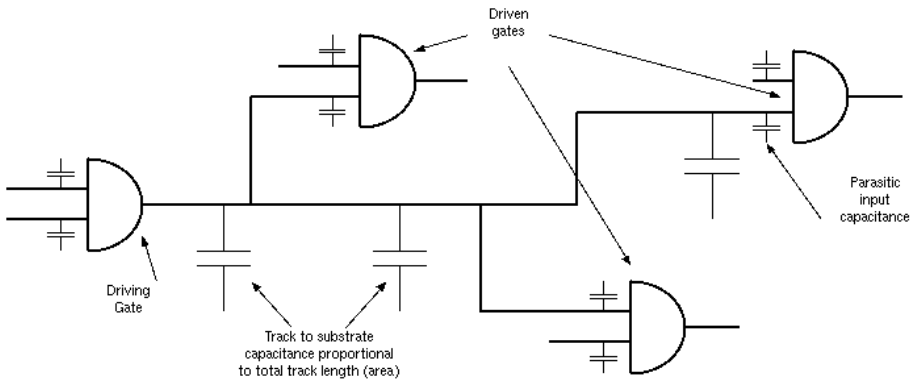


Figure 2.3: Logic net with tracking and input load capacitances illustrated.

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

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

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 4). Pre-layout and pre-synthesis we can predict net lengths from Rent's Rule and RTL-level heuristics.

Figure 2.3 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.

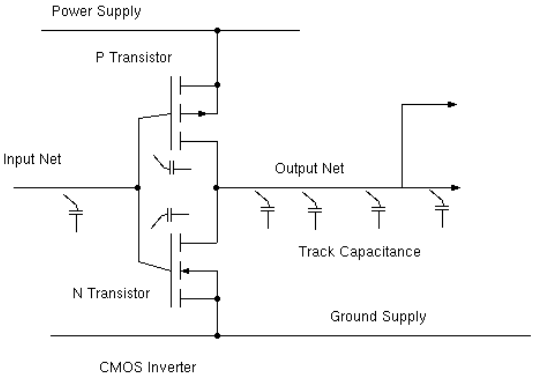
2.0.11 Detailed Power Model.

Power is measured in Watts and $P = V \times I = E \times f$

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

Early CMOS (VCC 5 volts): negligible static current, but today at VCC of 1.3 volts it's up to 30 percent of consumption.

Dynamic current = Short circuit current + Dynamic charge current.



Dynamic charge current computation:

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

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.

2.0.12 Dynamic Frequency and Voltage Scaling Example (DVFS)

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.

2.0.13 Deep submicron and Dark Silicon

Basic physical parameters for different technologies. Metal and polysilicon resistance are growing (bad) as we shrink.

Technology	0.25	0.18	0.15	0.13	0.09
Vdd (Volts)	2.5	1.8	1.5	1.2	1
No. of Metal layers	5	6	7	9	10
Dielectric constant	4.1	3.7	3.6	2.9	2.9
Resistivity - M1 (ohm-cm)	75	80	120	90	100
Resistivity - N+/Poly (Ohm/Ct)	7	8	13	11	20
Resistivity - via (Ohm/Ct)	4	7	9	1	1.5
Ca (aF/um)	15	11	11	15	14
Gate Delay (ps)	40	32	20	17	12

Figure 2.4: Technology Scaling (transistor on and off states getting closer!).

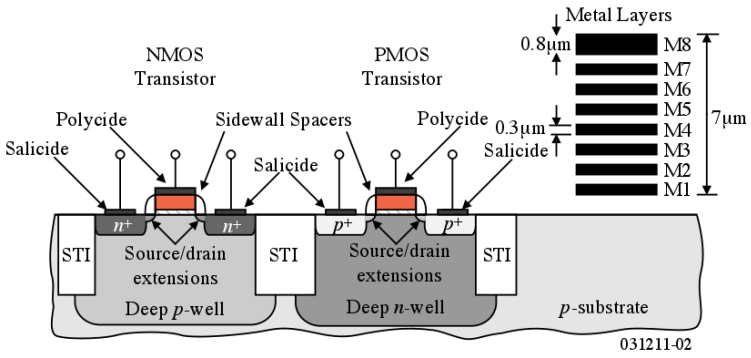


Figure 2.5: Schematic of transistor construction on silicon wafer.

2.0.14 Semi-Custom Design

Figure 3.4 shows a cell from the data book for a standard cell library. Such libraries are the modern equivalent of the 7400 range of logic gates and the silicon chip takes over from the breadboard (Figure 2.7). The illustrated 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.

Historically, there were two types of semi-custom devices:

- standard cell (for high volume)
- gate array (for volume less than 10,000 parts).

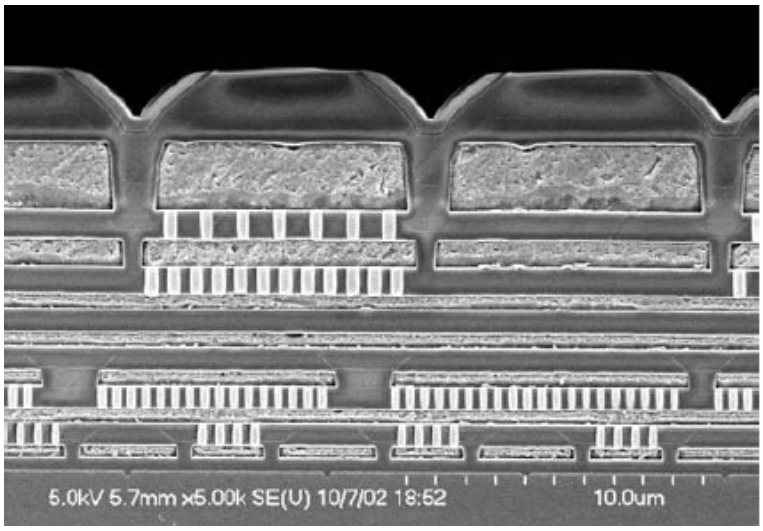


Figure 2.6: Cross section showing stacked wiring metal layers.

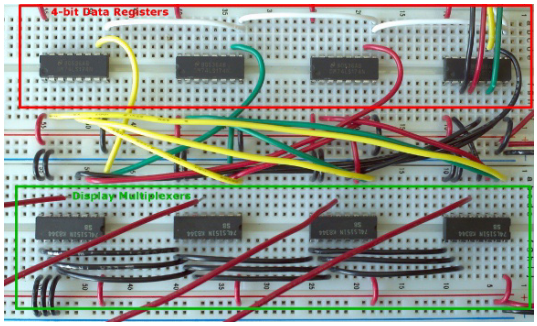


Figure 2.7: Discrete Logic Gates: Semicustom design puts them all on one die.

but now the mask-programmed gate array has been replaced with the field-programmed FPGA. FPGAs have also consumed a great portion of the previous standard cell market since the costs of custom masks cannot be amortised for production runse fewer than 50,000 or so.

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?

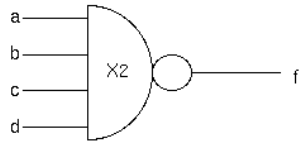
2.0.15 90 Nanometer Gate Length.

The mainstream VLSI technology in the period 2004-2008 was 90 nm. Now the industry is using 35-45 nanometer and smaller (but yield problems). Parameters from a 90 nanometer standard cell library:

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

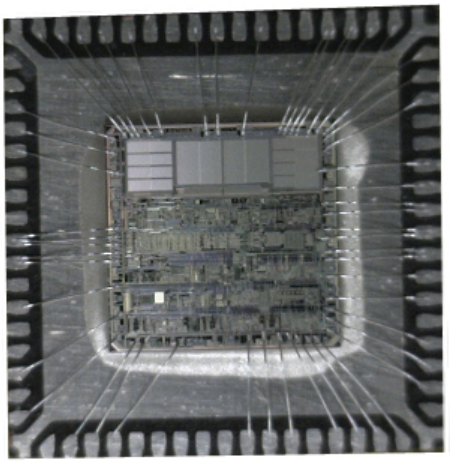
Min and Max delays depend upon temperature range, supply voltage, input edge speed and process spreads. The timing information is for guidance only. Accurate delays are used by the UDC.

CELL PARAMETERS : (One load unit = 49 fF)

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

Figure 2.8: Typical cell data sheet from a 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).

Moore's Law Transistor Count

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 than 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 is 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 two 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**.

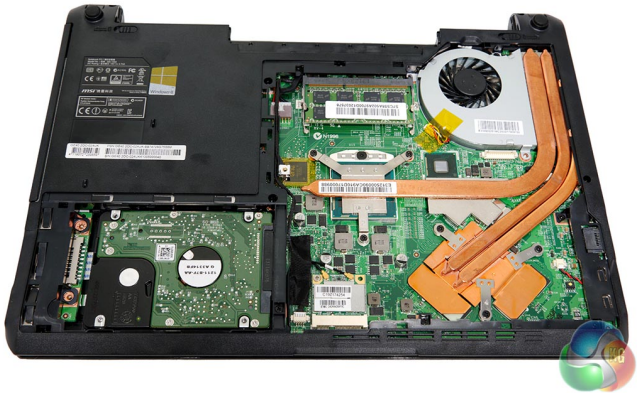


Figure 2.9: Thermal management heat pipes in a modern laptop.

2.0.16 Power Saving Techniques

We can save power by controlling power supplies and clock frequencies: Figure 2.10.

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

Figure 2.10: Terminology and Overview of Power Saving Techniques.

2.0.17 Save Power 1: 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 DESIGNLINE

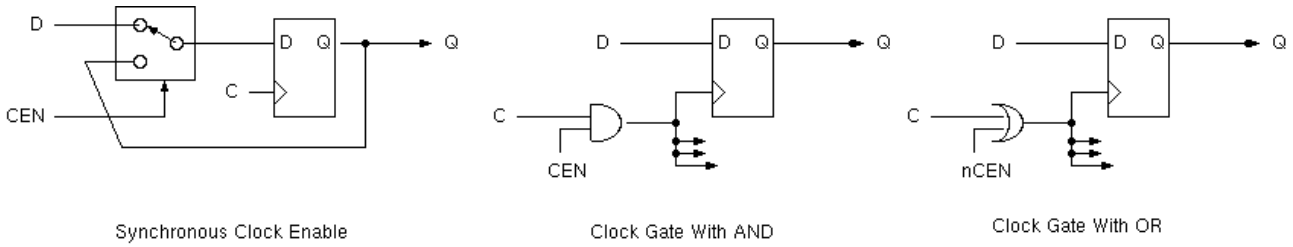


Figure 2.11: Clock enable using multiplexor, AND and OR gate.

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.

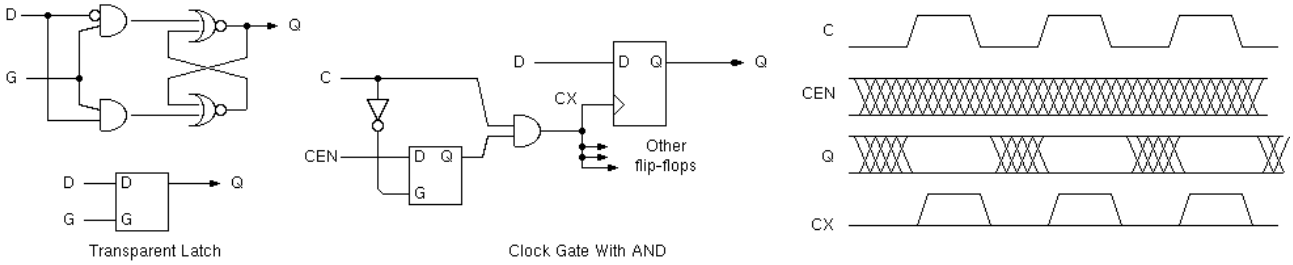


Figure 2.12: Illustrating a transparent latch and its use to suppress clock gating glitches.

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.

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.

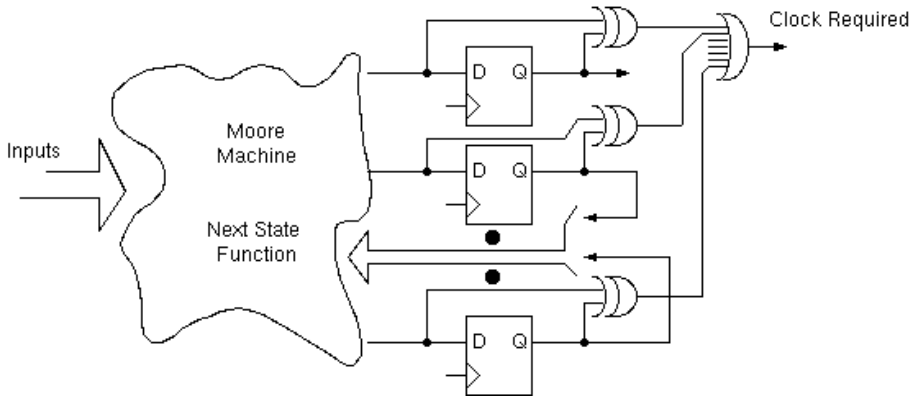


Figure 2.13: Using XOR gates to determine whether a clock edge would have any effect.

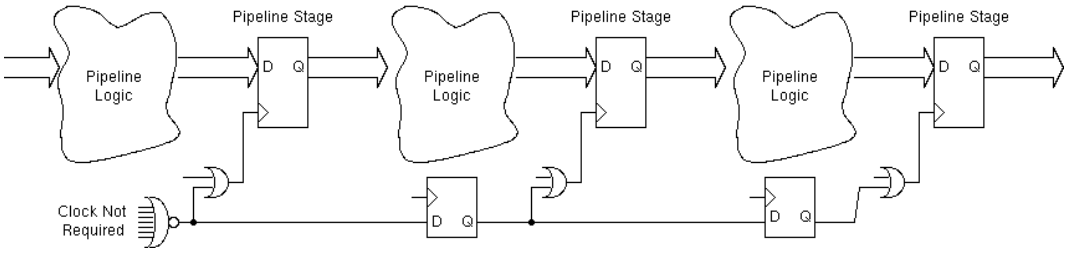
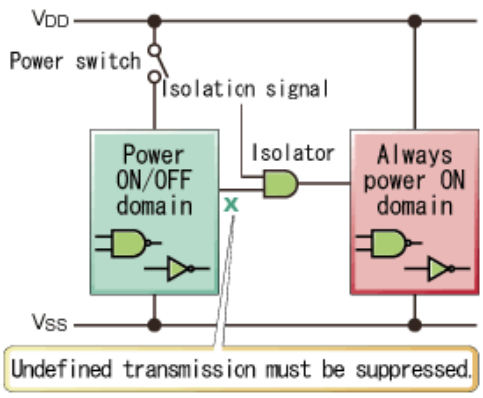
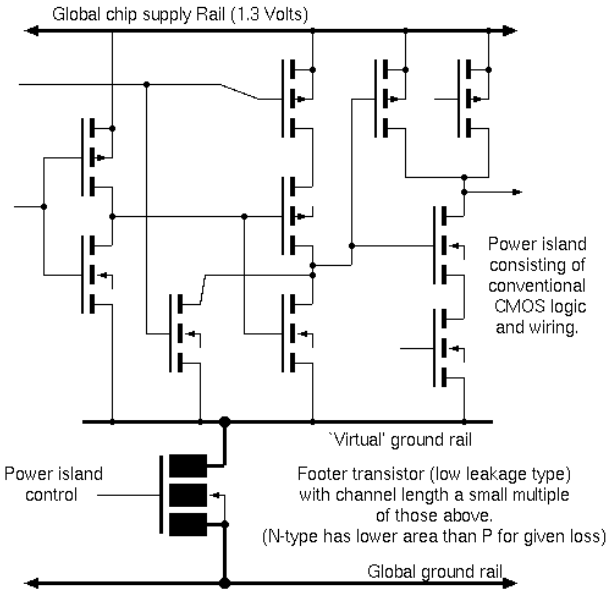


Figure 2.14: Clock needed computations forwarded down a pipeline.

2.0.18 Save Power 2: 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).





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).

2.0.19 Future Trends

Transistors are being made smaller and leakage current is going up. **Dark Silicon:** we can no longer turn on all of the chip: perhaps one tenth maximum for today's 22 nanometer chips. Even less in the future.

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.

SG 3 — 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 not in mainstream use (2015). 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

3.1 H/W to S/W Interfacing Techniques

The system is to be divided into some number of hardware and software blocks with appropriate means of interconnection. The primary ways of connecting H/W to S/W 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 addresses and device snoops data,
- DMA.

3.1.1 Conservation Cores Approach

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.

Dedicated hardware avoids instruction fetch overhead and is generally more power efficient. Analysis using Amdhal'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, with polled or interrupt-driven synchronisation with the main CPU.

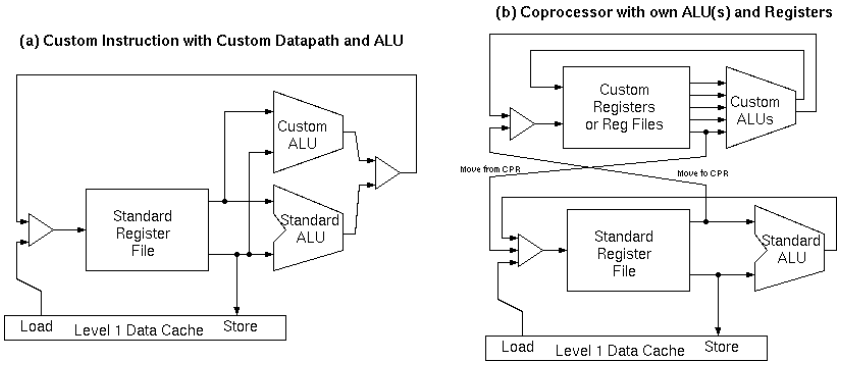


Figure 3.1: A custom ALU operation implemented in two similar ways: as a custom instruction or as a coprocessor.

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 hardware 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.

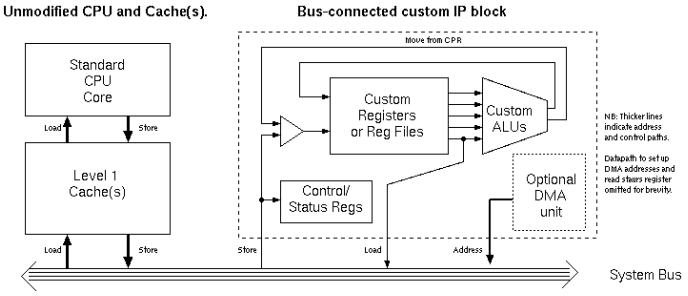


Figure 3.2: A custom function implemented as a peripheral IP block, with optional DMA (bus master) capability.

Whether to use the separate IP block really 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.

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 potential major trend for power conservation: sometimes called **conservation cores**.

3.1.2 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.

3.1.3 Chip Types and Classifications

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, ...).

Manufacturers can be classified as well:

1. Major chip makers such as IBM and Intel that design, manufacture and sell their chips (Integrated Device Manufacturers / IDM).
2. Fabless manufacturers such as NVIDIA and Xilinx that design and sell chips but outsource manufacturing to foundry companies.
3. 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 Limited

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

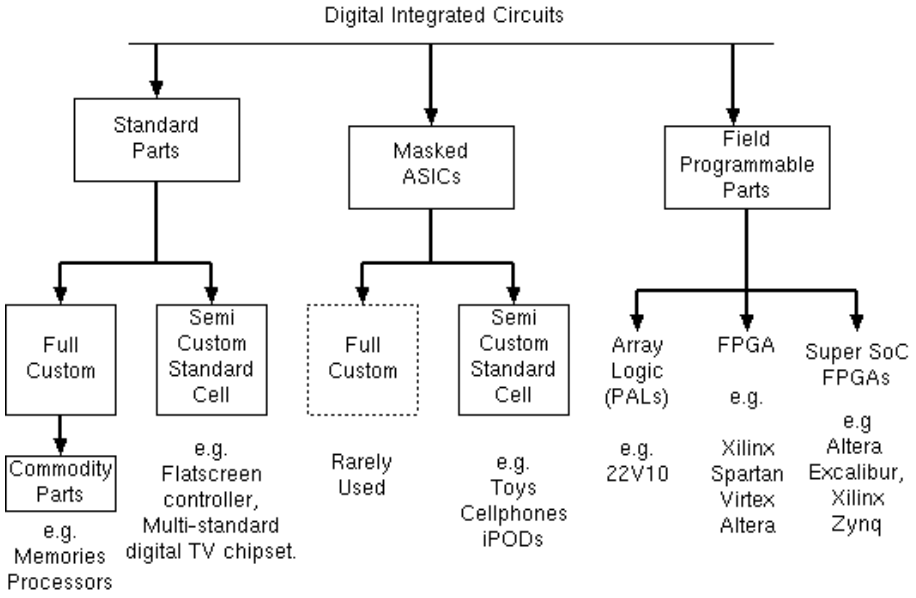


Figure 3.3: A taxonomy of integrated circuits.

Figure 3.3 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.

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), super FPGAs such as Zync are obliterating semi-custom masked ASICs for all but very-high-volume products. (When) Will FPGAs Kill ASICs?

3.1.4 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.

3.1.5 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.

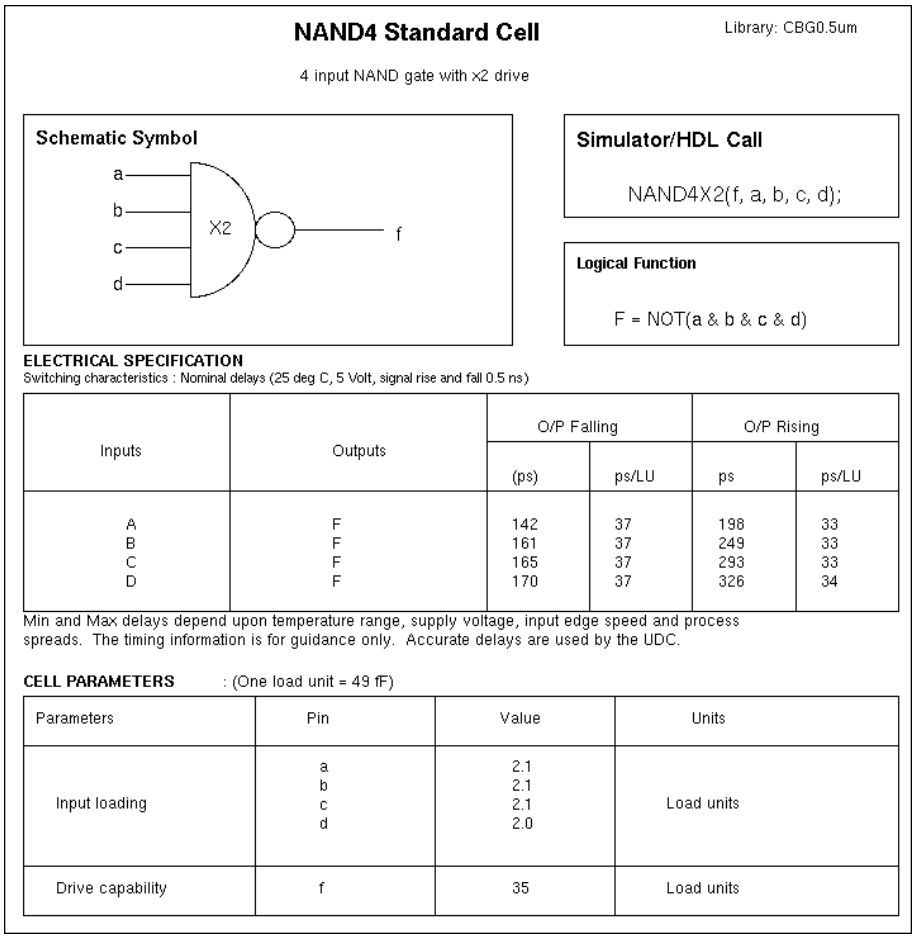
In semi-custom design, each cell has a fixed design and is repeated each time it is used, both within a chip and across many devices which have used the library. This simplifies design, but drive power of the cell is not optimised for each instance.

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

3.1.6 Semi-custom (cell-based) Design Approach

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 b c d	2.1 2.1 2.1 2.0	Load units
Drive capability	f	35	Load units

Figure 3.4: Typical cell data sheet from a standard cell library.

Figure 3.4 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 TRIAD

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.

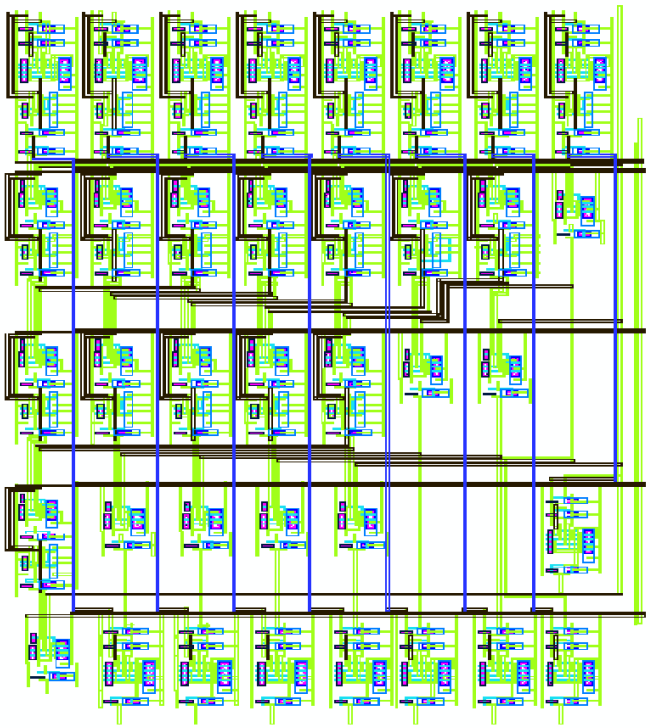


Figure 3.5: Standard cell layout for a Kogge-Stone adder. Taken from a student project (PDF on course web site).

3.1.7 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 TECH

Things 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.

3.1.8 Gate Arrays and Field-Programmable Logic.

Figure 3.6 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.

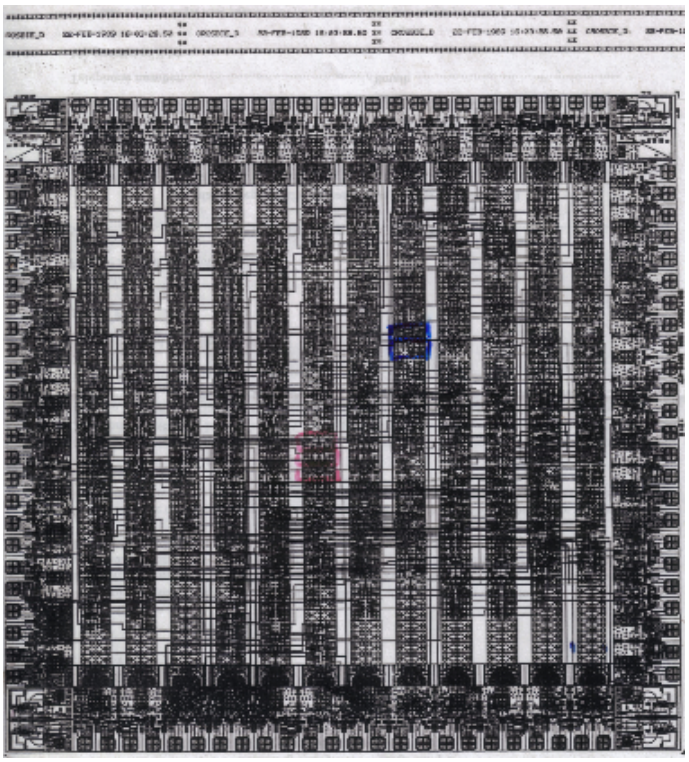


Figure 3.6: A Gate Array: (Greaves, Backbone Ring ECL Gate Array)

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.

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.

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. 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 FPGAs are now always used instead of masked gate arrays and are starting to kill ASCIs (see link above).

3.1.9 FPGA - Field Programmable Gate Array

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

Summary data for recent 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

65 nm technology, 6-input LUT, 64 bit DP RAMs.

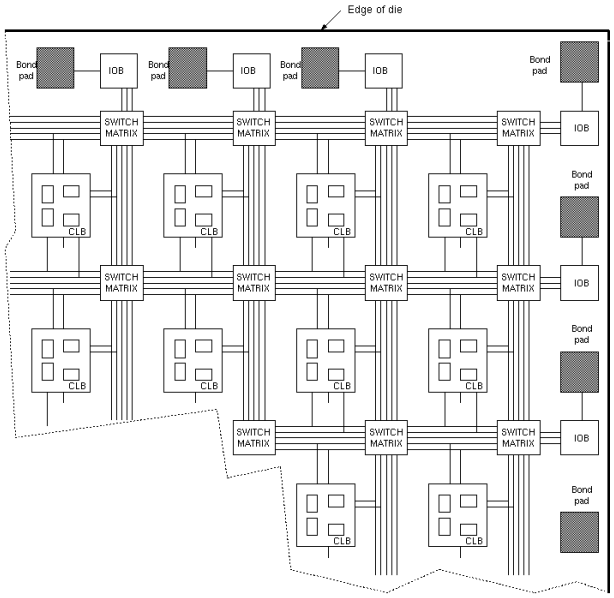
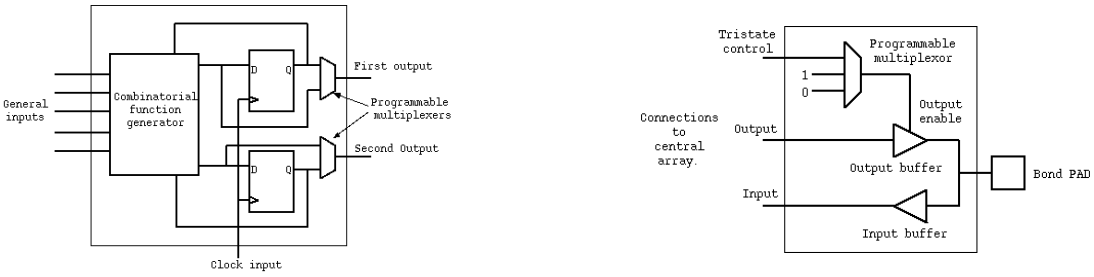


Figure 3.7: Field-programmable gate array structure, showing IO blocks around the edge, interconnection matrix blocks and configurable logic blocks. In recent parts, the regular structure is broken up by custom blocks, including RAMs and DSP ALUs.



An FPGA (field-programmable gate array) consists of an array of configurable logic blocks (CLBs), as shown in Figure 3.7. 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 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 hidden logic used only for configuration.

3.1.10 H/W versus S/W Design Partition Principles

The cost of developing an ASIC has to be compared with the cost of using an existing part. 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.

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 offerings: See Tensilica

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.

3.1.11 An old partitioning example: An external RS-232/POTS Modem.

Figure 3.8: A POTS modem.

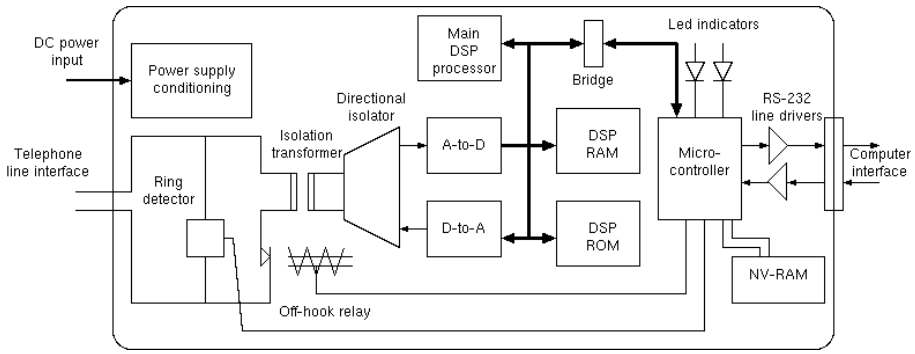


Figure 3.9: Example of a design partition — internal structure of the original modem.

Figure 3.9 shows the block diagram of a typical modem circa 1985. The illustrated device is an external modem, meaning that it sits in a box beside the computer and has an RS-232 serial connection to the computer. It also requires its own power supply.

The device contains a few analog components which behave broadly like a standard telephone, but most of it is digital. A relay is used to connect the device to the line and its contacts mirror the 'off-hook' switch which is part of every telephone. It connects a transformer across the line. The relay and transformer provide isolation of the computer ground signal from the line voltages. Similarly the ringing detector often uses a opto-coupler to provide isolation. *Clearly, these analog aspects of the design are particular to a modem and are designed by a telephone expert.*

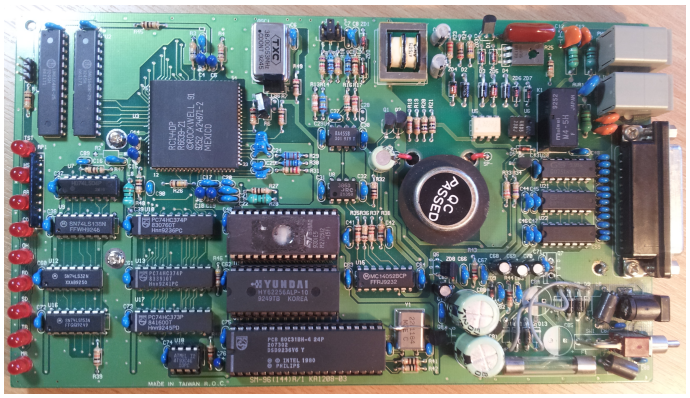


Figure 3.10: PCB of a similar modem - but offchip RAM and ROM for the 80C31 microcontroller.

Modems from the 1960's implemented everything in analog circuitry since microprocessors and DSP were not available. In 1985, two microprocessors were often used.

Note that the non-volatile RAM required (and still does) a special manufacturing processing step and so is not included as a resource on board the microcontroller. Similarly, the RS-232 drivers need to handle voltages of +/- 12 volts and so these cannot be included on chip without increasing the cost of the rest of the chip by using a fabrication process which can handle these voltages. The NV-RAM is used to store the owner's settings, such as whether to answer an incoming call and what baud rate to attempt a first connection, etc..

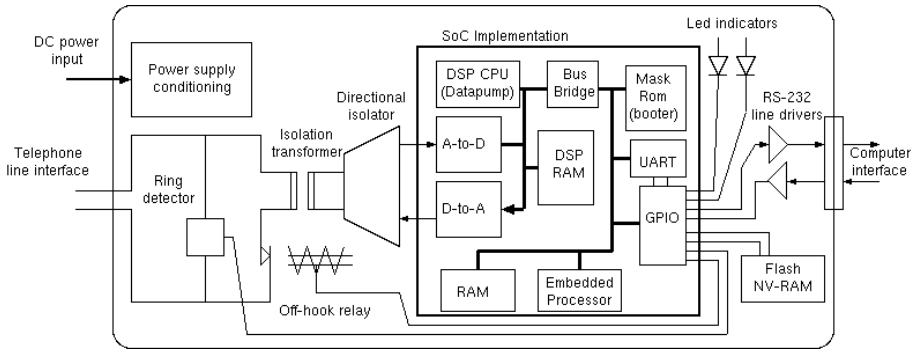


Figure 3.11: Typical structure of the modem product today (using a SoC approach).

A modern implementation would integrate all of the RAM, ROM, ADC and DAC and processors on a single SoC. The RS-232 remains off chip owing to 24 volt and negative supply voltages whereas the SoC itself may be run at 3.3 volts. The NV store is a large capacity Flash ROM device with low-bandwidth serial connection. At system boot, the main code for both processors is copied from the Flash to the two on-chip RAMS by the small, mask-programmed booter. Keeping the firmware in Flash allows the modem to be upgraded to correct bugs or encompass new communications standards.

GPIO is used for all of the digital I/O, with the UART transmit and receive paths being set up as special modes of two of the GPIO connections.

3.1.12 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

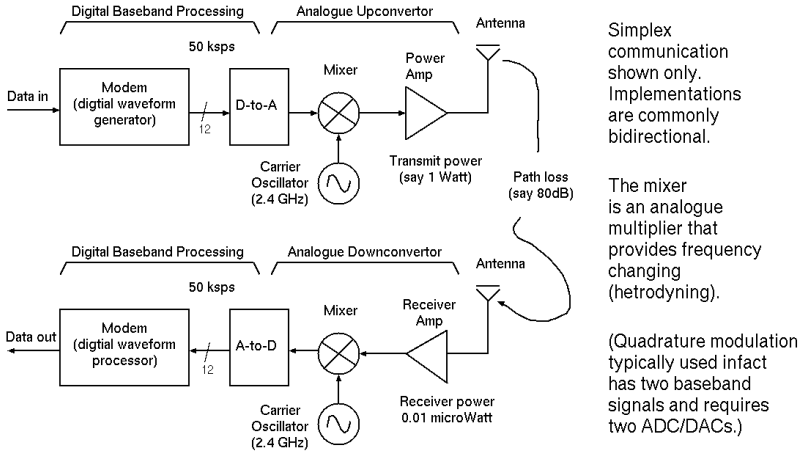


Figure 3.12: Typical structure of modern simplex radio link.

(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).

3.1.13 Partitioning example: A Bluetooth Module.

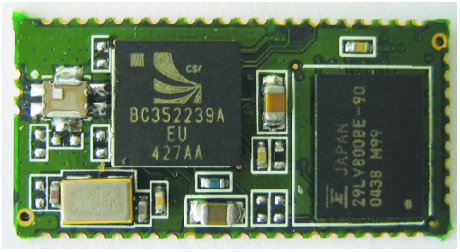


Figure 3.13: Broadcom (Cambridge Silicon Radio) Bluetooth Module circa 2000.

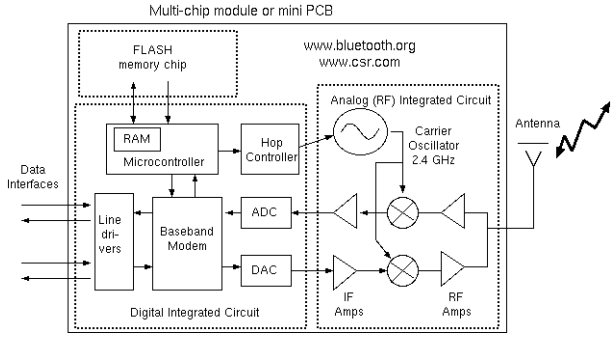


Figure 3.14: Example of a design partition — Block diagram of Bluetooth radio module (circa 2000).

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.

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.

3.1.14 ASIC costing.

The cost of a chip divides into two parts: NRE and per-device cost.

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

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

3.1.15 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

3.1.16 Xilinx Zynq Super FPGA

Xilinx Zynq-7000 Product Brief (PDF)

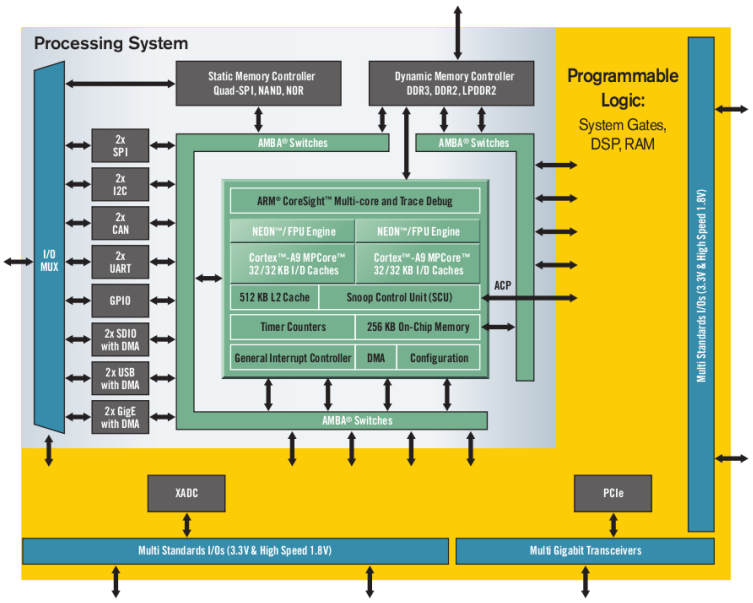


Figure 3.15: Xilinx Zynq 7000 Overview.

The high cost of ASIC masks now makes FPGA suitable for medium volume production. Super FPGAs, like Zynq emerge: the dark silicon trend means we can put all IP blocks on one chip provided we leave them mostly turned off. Xilinx Zynq solution has two ARM cores, all the standard IP blocks and an FPGA on one die. Flexible I/O routing means physical pads can be IP block bond outs, GPIOs or FPGA.

	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 3.16: Xilinx Zynq 7000 FPGA Resources.

	Zynq-7000 Product Table (Software View)			
Device Name	Z-7010	Z-7020	Z-7030	Z-7045
Part Number	XC7Z010	XC7Z020	XC7Z030	XC7Z045
Processing System — Processor Core	Dual ARM® Cortex™-A9 MPCore™ with CoreSight™			
Processor Extensions	NEON™ and Single/Double Precision Floating Point			
Maximum Frequency	800 MHz			
L1 Cache	32 KB Instruction, 32 KB Data per processor			
L2 Cache	512 KB			
On-Chip Memory	256 KB			
External Memory Support	DDR3, DDR2, LPDDR2			
External Static Memory Support	2x QSPI-SPI, NAND, NOR			
DMA Channels	8 (4 dedicated to Programmable Logic)			
Peripherals	2x USB 2.0 (OTG) w/DMA, 2x Tri-mode Gigabit Ethernet w/DMA, 2x SD/SDIO w/DMA, 2x UART (2), 2x CAN2.0B, 2x I2C, 2x SPI, 4x 32b GPIO			
Security	AES and SHA 256b for secure boot			
Peripherals and Static Memory Multiplexed I/O[1]	54			
Processing System to Programmable Logic Interface Ports (Primary Interfaces & Interrupts Only)	2x AXI 32b Master, 2x AXI 32b Slave, 4x AXI 64b/32b Memory AXI 64b ACP 16 Interrupts			
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)
Extensible Block RAM (# 36 Kb Blocks)	240KB (60)	560KB (140)	1,060KB (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	AES and SHA 256b for secure configuration			
Multi-Standards 3.3V I/O[2]	100	200	250	350
Serial Transceivers[2]	—	—	4	16

Figure 3.17: Xilinx Zynq 7000 ARM Cores, RAM, DRAM and DMA summary.

SG 4 — Verilog RTL: Modules, Protocols and Interfaces

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 have no physical manifestation.

In a modern design flow, the protocol at an interface is specified once in a master file that is imported for the synthesis of each module that sports it.

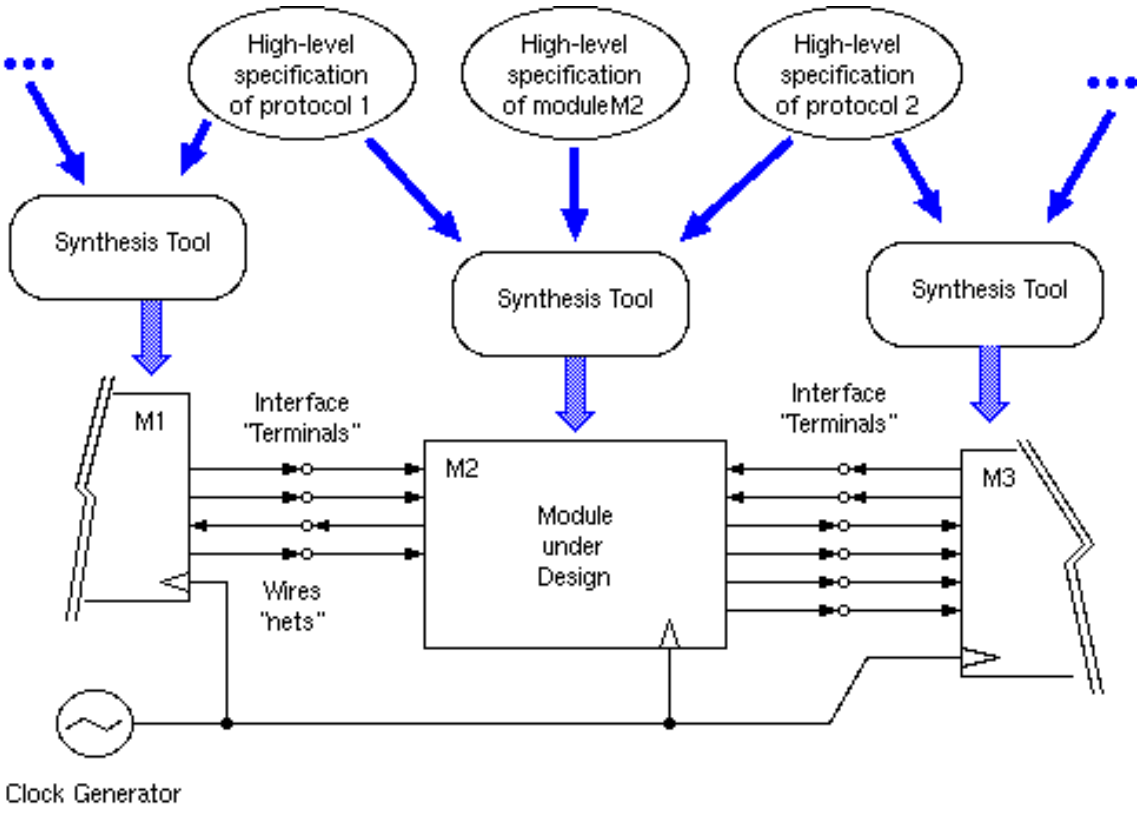


Figure 4.1: Generic (net-level) Module Interconnection Using Protocols and Interfaces.

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 basic 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)

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 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.

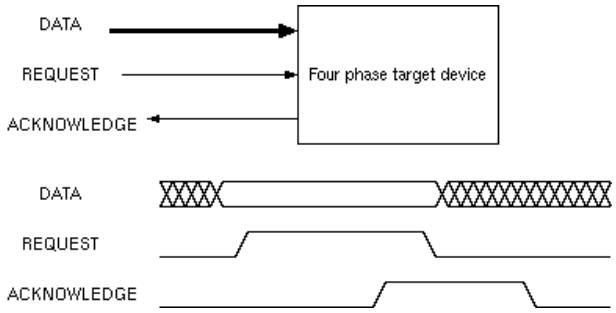


Figure 4.2: Timing diagram for an asynchronous, four-phase handshake.

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 IO (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)

A more complex example is the LocalLink protocol from Xilinx. This is a synchronous packet proctocol (compare with the asynchronous four-phase handshake just described).

Like the four-phase handshake, LocalLink has contra-flowing request and acknowledge signals. But data is not

qualified by a request transition: instead it is qualified as valid on **any positive clock edge where both request and acknowledge are asserted**. 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
```

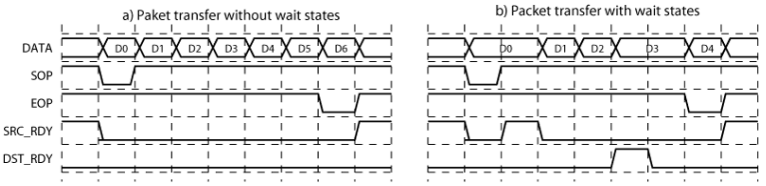


Figure 4.3: Timing diagram for the synchronous LocalLink protocol.

Start and end of frame signals delimit the packets. All control signals are active low (denoted with the underscore n suffix).

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 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.

4.2.1 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
```

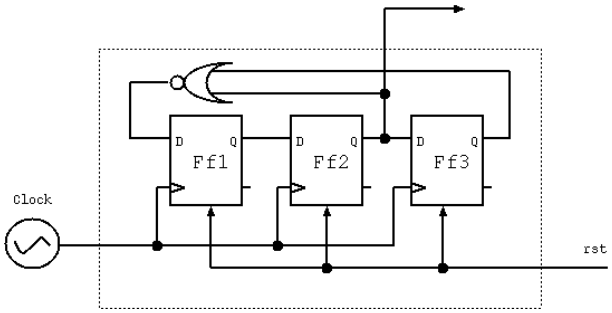


Figure 4.4: The circuit described by our structural example (a divide-by-five, synchronous counter).

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

All hardware description languages and RTLs contain some sort of **generate statement**. A generate statement is an itereative construct that is executed at compile time to generate multiple instances of a component. In Bluespec this is a complete, higher-order functional language, but in SystemVerilog we use the following:

```
reg[3:0] values[0:4] = {5, 6, 7, 8, 9};

 generate
   genvar i;
   for (i=0; i < 5; i++)  begin: M1
     MUT mut(
     .out,
     .in(values[i]),
     .clk
     );
   end
 endgenerate
```

Heirarchic Netlist

```
module MOD1(output b,  input a);
  wire c;

  INV inv1(c, a);
  MODX modx1(b, c);
endmodule

module MOD2(output q, input s, input r);
  wire c;

  INV inv2(c, s);
  MODY mody1(q, c, r);
endmodule

module MODTOP(output rr, input aa, input bb);
    wire l, m;

  MOD1 m(l, aa);
  MOD1 n(m, bb);
  MOD2 o(rr, l, m);
endmodule
```

Equivalent Flattened Netlist

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

  INV m_inv1(m_c, aa);
  INV n_inv1(n_c, bb);
  INV o_inv2(o_c, l);
  MODX m_modx1(m_c, l);
  MODX n_modx1(n_c, m);
  MODY o_mody1(rr, o_c, m);

endmodule
```

For many designs the flattened netlist is often bigger than the hierarchic netlist owing to multiple instances of the same component. Here it was smaller.

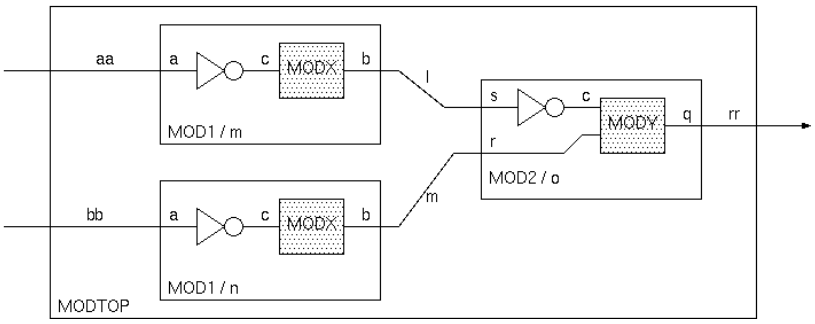


Figure 4.5: Example RTL fragment, before and after flattening.

Figure 4.5 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;
```

- Order of continuous assignments is un-important,
- Loop free, otherwise: parasitic 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;
```

```
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);  // Note handout uses older syntax here

  reg [3:0] count, oldcount;

  // Add a four bit decimal value of one to count
  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'). .xi Each register is assigned in exactly one clock domain. RTL synthesis does .xi 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

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

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

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

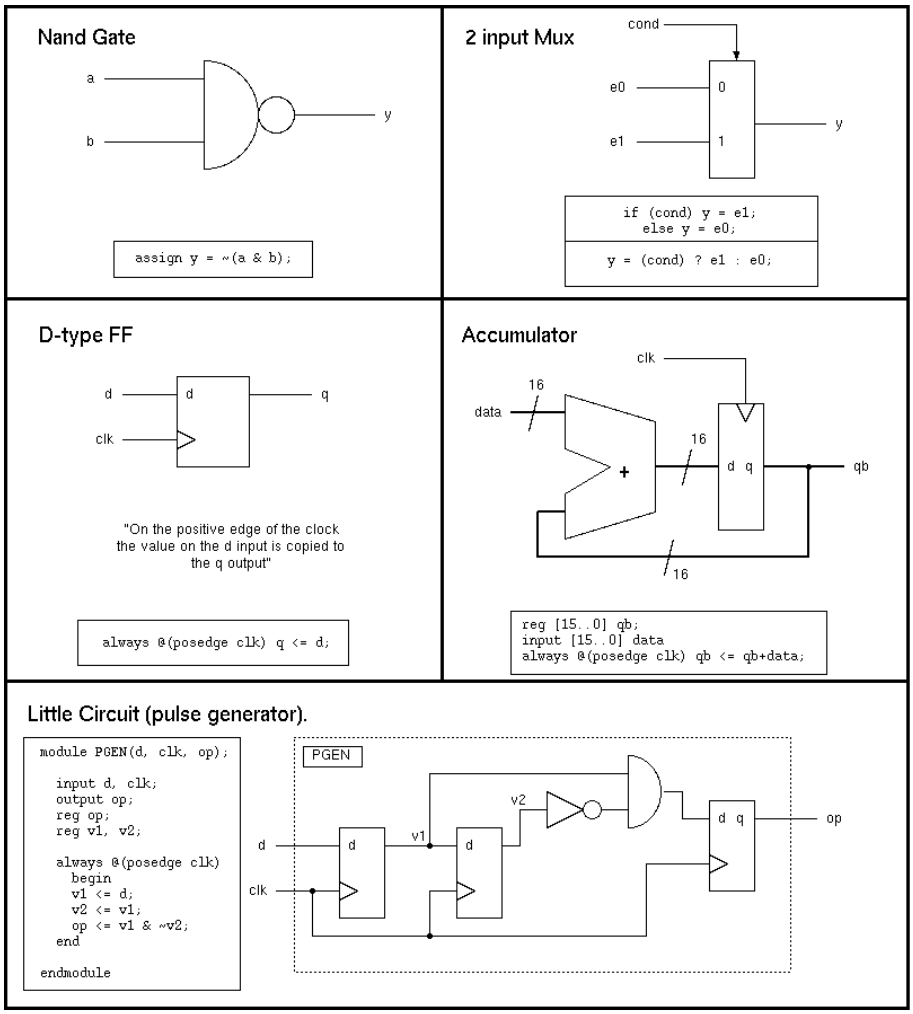


Figure 4.6: Elementary Synthesisable Verilog Constructs

Figure 4.6 shows synthesisable Verilog fragments as well as the circuits typically generated.

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. .xi We should certainly leave out the test-bench wrapper when synthesising and .xi we typically want to synthesise each major component separately.

4.2.2 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 so-called structural generation statements 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 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.

Additional notes:

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
```

Can 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.

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;
```

There are three ML fragments on the course web site that demonstrate each step of this recipe: 1: pure conversion 2: bit-blasting 3: gate building *(The details of the algorithms and being able to reproduce them is not examinable but being able to draw the gate-level circuit for a few lines of RTL is examinable).*

4.2.3 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 req [3:0] q;

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

How many bits of PC are needed ? Is conditional event control synthesisable ? Does the output 'q' ever take on the value 4 ?

As a second non-synthesisable example, consider the dual-edge-triggered flip-flop in Figure 4.7.

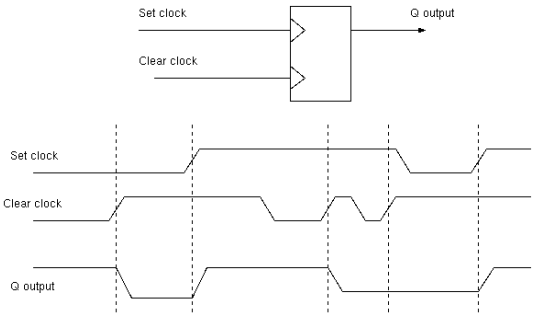


Figure 4.7: Schematic symbol and timing diagram for an edge-triggered RS flop.

```
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.

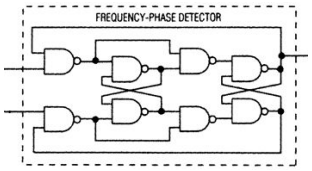


Figure 4.8: Hand-crafted circuit for the edge-triggered RS flop used in practice.

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

```
// Typical RTL testbench contents:
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
```

4.2.4 Further 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,
- Area use,
- Performance,
- Testability.

(The basic algorithm in the additional material does not consider any guiding 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! Can re-encode state (see later).

4.2.5 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 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.

4.3 Alternatives to RTL

Higher-level entry forms are ideally needed, perhaps schedulling within a thread at compile-time and between threads at run time ?

High-level Synthesis (HLS) essentially converts software to hardware. Classically it take one thread and a fixed body of code and it

- unwinds inner loops by some factor,
- generates a custom **datapath** containing registers, RAMs and ALUs
- and a custom **sequencer** that implements an efficient, static **schedule**

that achieves the same behaviour as the original program. It will generally deploy state re-encoding and re-pipelining to meet timing closure and power budgets.

Greaves wrote an HLS program in around 1990 that was commercially licensed: CTOV Bubble SorterExample. These ideas are only now being seriously considered by industry, with all major tools providers now offering a

C-to-gates compiler. LegUp from Toronto is a modern equivalent. The Kiwi Compiler uses similar approaches for acceleration of big data algorithms expressed in concurrent CSharp on FPGA.

For the future, the following two look promising: Chisel: Constructing Hardware in a Scala-Embedded Language and HardCaml - Register Transfer Level Hardware Design in OCaml

4.3.1 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 operation has a guard predicate: says when it CAN be run.

Operations are grouped into 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. Operations have the expectation they WILL be run (fairness). A compiler can direct scheduling decisions to span various power/performance implementations for a given program.

- A Bluespec design is expressed as a list of declarative rules,
- Shared variables are not used
- All comunication to and from registers, FIFOs and user modules is via transactional/blocking 'method calls' for which argument and handshake wires are synthesised according to a global ready/enable protocol,
- Rules are allocated a static schedule at compile time and some that can never fire are reported,
- The current strict mapping to clock cycles (time/space folding) might be relaxed by future compilation strategies.
- The wiring pattern of the whole design is generated using an embedded functional language (like Lava HDL).

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

LINK: Small Examples Toy BSV Compiler (DJG)

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

```
module mkTb1 (Empty);

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

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

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

endmodule: mkTb1
```

Second example uses a pipeline object that could have aribtrary delay. Sending process is blocked by implied handshaking wires (hence less typing than Verilog) and in the future would allow the programmer or the compiler to retime the implementation of the pipe component.

```
module mkTb2 (Empty);

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

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

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

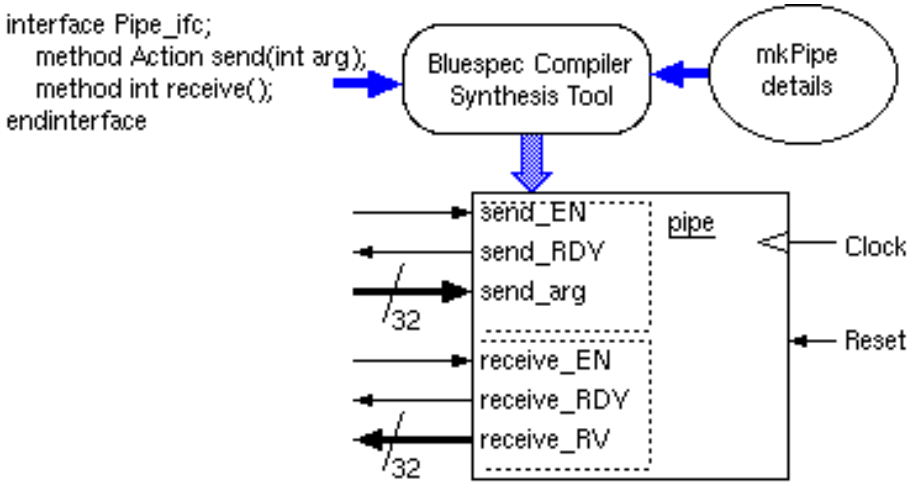


Figure 4.9: Synthesis of the 'pipe' Bluespec component with handshake nets.

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

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 simulation, and
- (EDS) event-driven simulation.

Finite-difference simulation is used for analogue and fluid-flow systems. It is rarely used in SoC design (just for low-level electrical propagation and cross-talk 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:

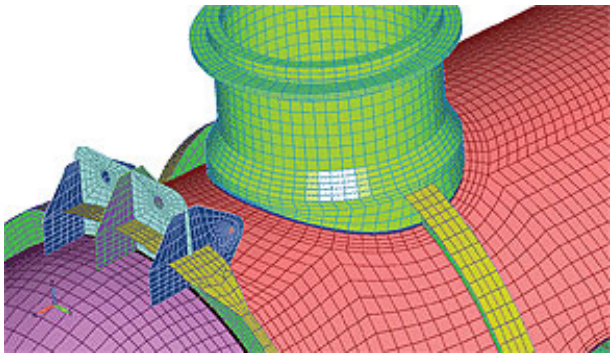


Figure 4.10: Finite Element Grid

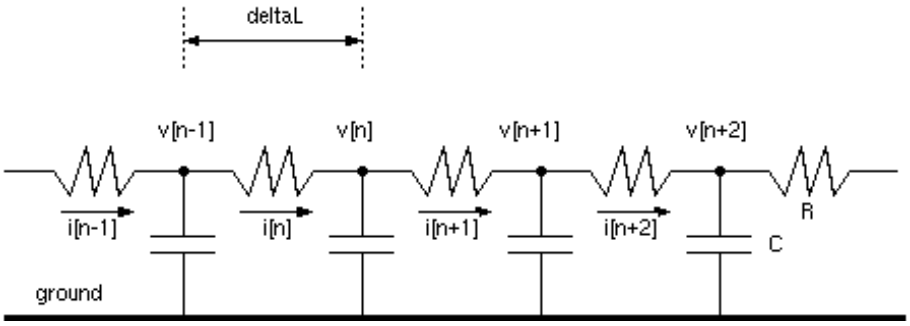


Figure 4.11: Baseline finite-difference model for bidirectional propagation in one dimension.

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

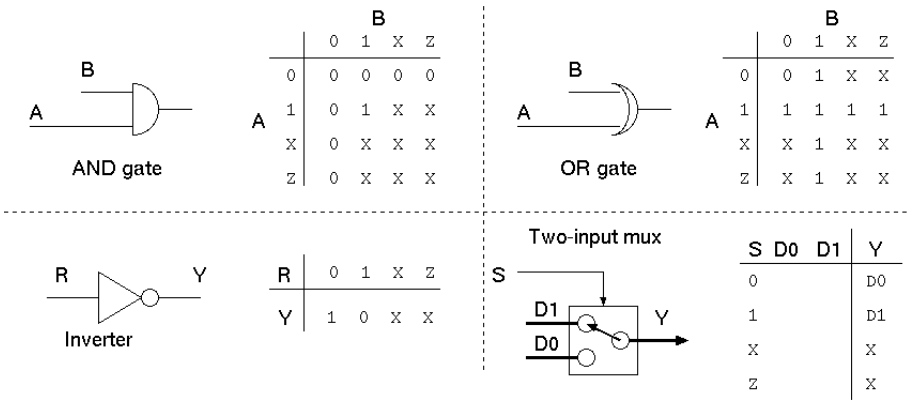
Generally use a 2D or 3D grid for fluid modelling: 1D ok for electronics. Typically want to model both resistance and inductance for electrical system. When modelling inductance instead of resistance, then need a '+=' in the $i[n]$ equation. When non-linear components are present (e.g. diodes and FETs),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

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 design specification, the letter 'x' is also used to denote 'dont-care', which allows efficient logic minimisation. In Verilog, the letter 'x' means uncertain during simulation and 'dont-care' during logic synthesis.



A \ B	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

A \ B	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

R	0	1	X	Z
Y	1	0	X	X

S	D0	D1	Y
0			D0
1			D1
X			X
Z			X

Figure 4.12: Illustratring the four-value logic level encoding for common gates.

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

4.4.2 Event Driven Simulation

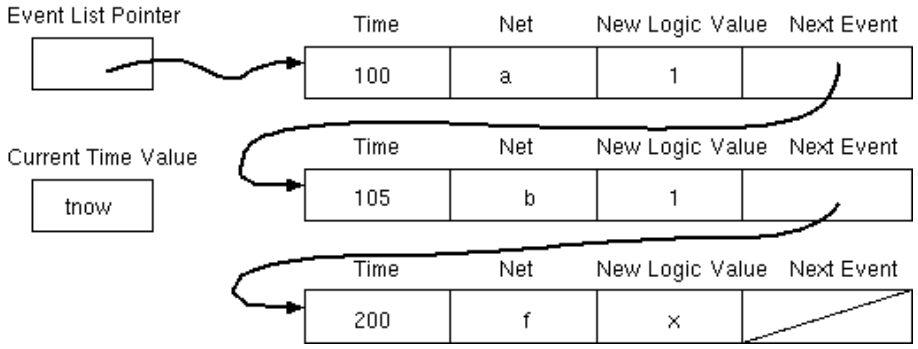


Figure 4.13: Event queue, linked list, sorted in ascending temporal order.

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

```
// 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.

We will cover two variations on the basic EDS algorithm: intertial delay and delta cycles.

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 Inertial and Transport Delay

Consider a simple 'NOR' gate model with 250 picosecond delay. It has two inputs, and the behavioural code inside the model will be something like (in SystemC-like syntax, covered later)

```
SC_MODULE(NOR2)
{  sc_in < bool > i1, i2; sc_out < bool > y;
   void behaviour()
   {  y.write(!(i1.read() || i2.read()), SC_TIME(250, SC_PS));
   }
   SC_CTOR(NOR2) { SC_METHOD(behaviour); sensitive << i1 << i2;
}
```

The above model is run when either of its inputs change and it causes a new event to be placed in the event queue 250 picoseconds later. This will result in a pure **transport delay**, because multiple changes on the input within 250 picoseconds will potentially result in multiple changes on the output that time later. This is unrealistic, a NOR gate made of transistors will not respond to rapid changes on its input, and only decisively change its output when the inputs have been stable for 250 picoseconds. In other words, it exhibits inertia. To model **inertial delay**, the event queue insert function must scan for any existing schedulled changes before the one about to be inserted and delete them. This involves little overhead since we are scanning down the event queue anyway.

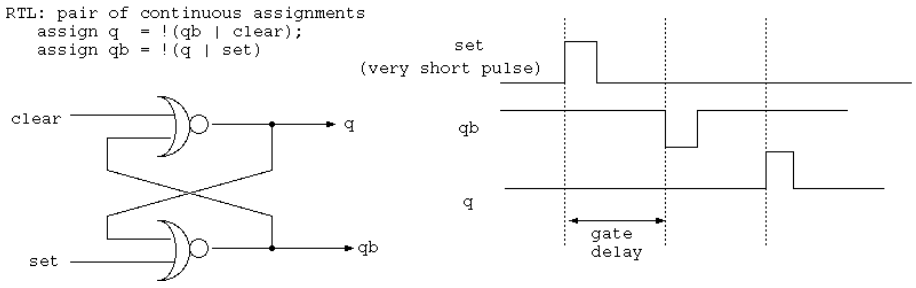


Figure 4.14: RS-latch: behaviour of runt pulse when modelling with transport delay.

Consider the behaviour of the above RS-latch when a very short (runt) pulse or glitch tries to set it. What will it do with transport models?: the runt pulse will circulate indefinitely. What will it do with inertial models?: ignore the glitch.

4.4.4 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 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.

4.4.5 Static Timing Analyser Tool

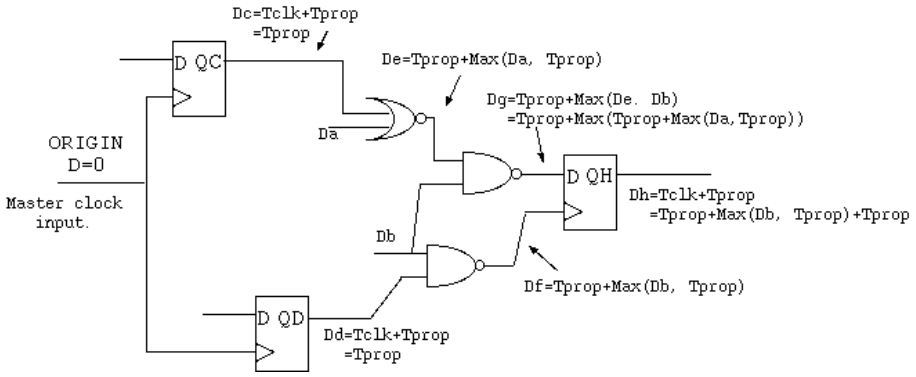


Figure 4.15: An example circuit with static timing annotations

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.

An alternative to simulation is static analysis. For delay, finding the critical path, we can use a static timing analyser.

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.

4.5 Hazards

Definitions (some authors vary slightly):

- **Data hazard** - when an operand's address is not yet computed or has not arrived in time for use,
- **WaW hazard** - write-after-write: the second write must occur after the first otherwise its result is lost,
- **RaW or WaR hazard** - write and read of a location are accidentally permuted,

- **Control hazard** - when it is not yet clear whether an operation should be performed,
- **Alias hazard** - we do not know if two array subscripts are equal,
- **Structural hazard** - insufficient physical resources to do everything at once.

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 *non-fully pipelined* component cannot start a new operation on every clock cycle. Instead it has handshake wires that start it and inform the client logic when it is ready.

Synchronous RAMs 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:

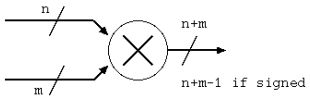
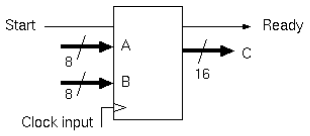
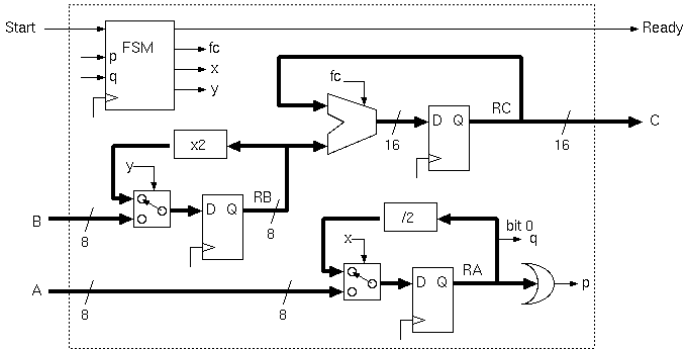


Figure 4.16: Multiplier schematic symbol.

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 illustrateddatapath
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.

Exercise: Write out the complete design, including sequencer, for the above multiplier, or that of Booth, or a long division unit, in Verilog or SystemC.

For today's VLSI, for 32 bits, a fixed latency multiplier is typically used. This will have 2 or 3 clock cycles delay and be fully pipelined.

Fully-pipelined logic with fixed latency is easy to incorporate into static schedules, either by hand or with automated HLS tools.

Recent email from Dr Mullins: At a clock 500MHz, I see something like 13.5pJ for a 32-bit multiply (2-stage pipeline)in a 40nm low power (LP) silicon process. Its area is 8750um.

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.

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

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).

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 HLS). 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:

```
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 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 shall see this at the gate level later on). 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).

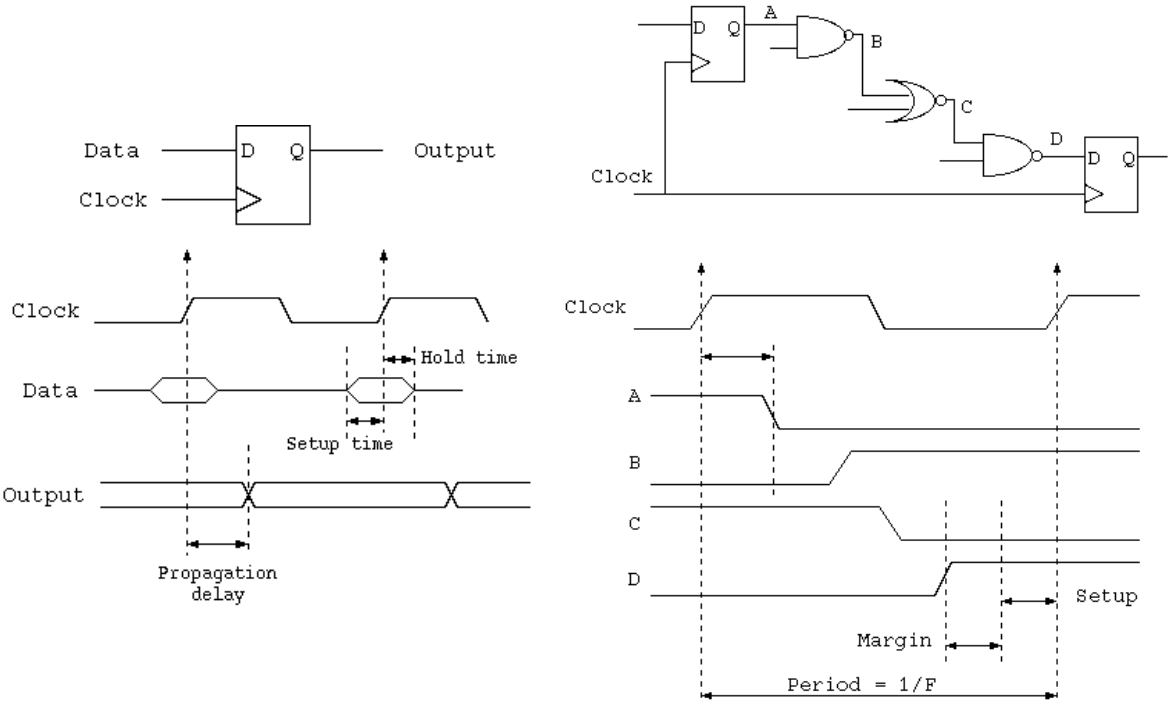
```
LOOPED (time) option:                | UNWOUND (space) option:
                                     |
for (i=0; i < 3 and i < limit; i++)  |  if (0 < limit) sum += data[0] * coef[j];
   sum += data[i] * coef[i+j];       |  if (1 < limit) sum += data[1] * coef[1+j];
                              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

4.6.1 Critical Path Timing Delay

Meeting **timing closure** is the process of manipulating a design to meet its target 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.

Pipelining is a commonly-used technique to boot 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. 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:

```
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

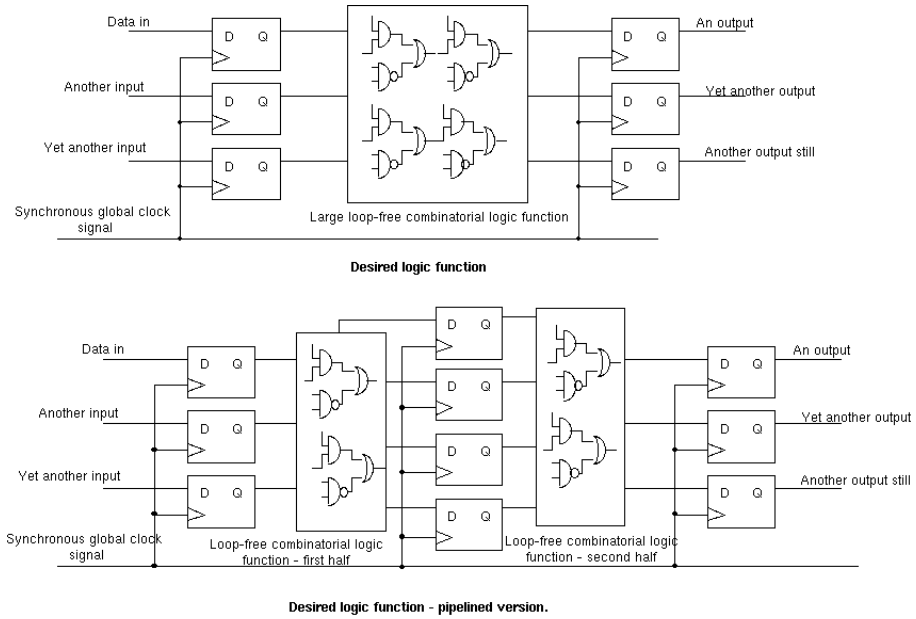


Figure 4.17: A circuit before and after insertion of an additional pipeline stage.

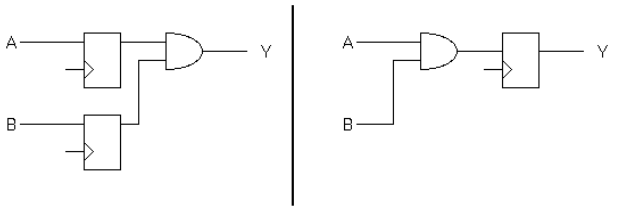


Figure 4.18: Flip-flop migration: two circuits of identical behaviour, but different state encoding.

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)(see later) 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, 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 introduce one-hot and gray encoding, or invert for reset as preset.

4.6.2 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 2.0.10).

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 1.3.4).

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

SG 5 — Formal Methods and Assertion-Based Design

Topics: Declarative expression. Temporal Logic. PSL. Assertion Synthesis to H/W Monitors. Stimulus generation.

5.1 Assertions

Declarative programming involves writing assertions that hold for all time. For instance, on an indicator panel *never is light A on at the same time as light B.*

Assertion-based design (ABD) is an approach that encourages writing assertions as early as possible, preferably *before* coding/implementation starts.

- Writing assertions at design capture time *before* detailed coding starts.
- Writing further assertions as coding progresses.
- Structuring testing around assertions.

Assertions are (conjunctions of):

- Imperative (aka immediate) safety checks (like `assert.h` in C++ and `expect` in SystemVerilog)
- Coverage checks (log that flow of control has passed a point or a property held).
- Declarative safety properties, that always hold, such as 'Never are both the inner and outer door of the airlock open at once unless we are on the ground'. Safety properties normally use the keywords **never** or **always**.
- Liveness and deadlock properties (also declarative), such as 'If the emergency button is pressed, eventually at least one of the doors will be unlocked.' Liveness properties normally use keywords such as **eventually** or **it will always be possible to**.

All four can be proved by formal techniques such as pen and paper, theorem provers and model checkers. *Dynamic validation* is simulation while checking properties. This can sometimes find safety violations and sometimes find deadlock but it cannot prove the liveness.

Assertions can be imported from previous designs or other parts of the same design for global consistency. Formal proof shows up corner case problems not encountered in simulation. A formally-verified result may be *required* by the customer.

5.1.1 Validation using Simulation

The alternative to formal verification is validation using extensive simulation and overnight testing of the day's work using **regression testing**.

Tests can be **unit** tests or larger subsystems or complete system (H/W + S/W).

Can either write a RTL or ESL yes/no automaton as part of the test bench, or one can spool the outputs to file and **diff** against **golden** with PERL script.

Downfall of simulation: it's non-exhaustive and time consuming.

ABD benefits (and challenges):

- Completeness (how to define this?)
- Scalability (tools limited in practice?),
- Rare corner situations (unusual conjunctions of events) are covered.

But: Simulations

- are needed for performance analysis and general design confidence,
- can generate some production **test vectors**.
- can be partly formal: using bus monitors for dynamic validation and Specman/VERA constrained pattern generators for stimulus.

Simulation is effective at finding many early bugs in a design. It can sometimes find safety violations and sometimes find deadlock but it cannot prove liveness.

Once the early, low-hanging bugs are fixed, formal proof can be more effective at finding the remainder. These tend to lurk in unusual corner cases, where particular alignment or conjunction of conditions is not handled correctly.

If a bug has a one in ten million chance of being found by simulation, then it will likely be missed, since fewer than that number clock cycles might typically be simulated in any run. However, given a clock frequency of just 10 MHz, the bug might show up in the real hardware in one second!

Simulation is generally easier to understand. Simulation gives performance results. Simulation can give a golden output that can be compared against a stored result to give a pass/fail result. A large collection of golden outputs is normally built up and the current version of the design is compared against them every night to spot **regressions**.

Simulation **test coverage** is expressed as a percentage. Given any set of simulations, only a certain subset of the states will be entered. Only a certain subset of the possible state-to-state transitions will be executed. Only a certain number of the disjuncts to the guard to an IF statement may hold. Only a certain number of paths through the block-structured behavioural RTL may be taken. Medical, defense and aerospace generally require much higher percentage coverage than commercial products.

There are many ways of defining coverage: for instance do we have to know the reachable state space before defining the state space coverage, or can we use all possible states as the denominator in the fraction? In general software, a common coverage metric is the percentage of lines of code that are executed.

Scaling of formal checking is a practical problem: today's tools certainly cannot check a complete SoC in one pass. An incremental approach based around individual sub-systems is needed.

5.1.2 Formally Synthesised Bus Monitor

A bus monitor is a typical example of dynamic validation: it is a checker that flags protocol violations:

- safety violations are indicated straightaway,
- for a liveness property the monitor can indicate whether it has been tested at least once and also whether there is a pending antecedant that is yet to be satisfied.

For implementation in silicon, or if we are using an old simulator (e.g. a Verilog interpreter) that does not provide PSL or other temporal logic, the assertions can be compiled to an RTL checker automaton.

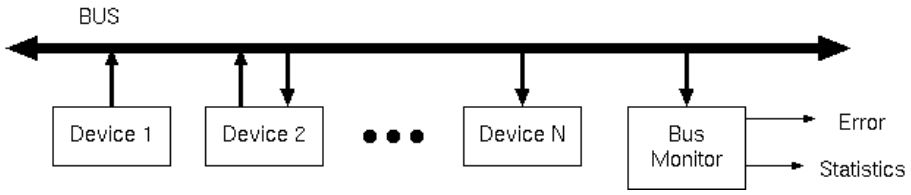


Figure 5.1: Dynamic validation: Monitoring bus operation with a hardware monitor.

A bus monitor connects to the net-level bus in RTL or silicon. (TLM formal monitoring is also being developed.)

The monitor can keep statistics as well as detect protocol violations.

Example of checker synthesis from a formal spec: www.cl.cam.ac.uk/research/srg/han/hprls/orangepath/transactors and Bus Monitors

5.1.3 Is a formal specification complete ?

Additional notes:

Is a formal specification complete ?

- Does it fully-define an actual implementation (this is overly restrictive) ?
- Does it exactly prescribe all allowable, observable behaviours ?

By 'formal' we mean a machine-readable description of what is correct or incorrect behaviour. A **complete** specification might describe all allowable behaviours and prohibit all remaining behaviours, but most formal definitions today are not complete in this sense. For instance, a definition that consists of a list of safety assertions and a few liveness assertions might still allow all sorts of behaviours that the designer knows are wrong. He can go on adding more assertions, but when does he stop ?

One might define a 'complete specification' as one that describes all observable behaviours. Such a specification does not restrict or prescribe the internal implementation in black box terms since this is not observable.

When evaluating an assertion-based test program for an IP block, we can compute assertion coverage in many ways: e.g. What percentage of rule disjuncts held as dominators (on their own) ? Or, e.g. What percentage of reachable state space was spanned?

5.1.4 Assertion forms: State/Path, Concrete/Symbolic.

Many assertions are over **concrete state**. For instance *'Never is light A off when light B is on'* . Other assertions need to refer to **symbolic values**. For instance *'The value in register X is always less than the value in register Y'* .

State properties describe the current state only. For instance '*Light A is off and light B is on*'. **Path properties** relate successive state properties to each other. For instance '*light A always goes off before light B comes on* '.

We shall see PSL requires the symbolic values be embedded in the bottommost 'modelling layer' and that its temporal layer cannot deal with symbolic values. For instance, we cannot write '$\{A(x); B(y)\} \mid => \{C(x, y)\}$'.

(Note: the internal representation used by a checker tool for a concrete property can commonly use a symbolic encoding, such as a BDD, to handle an exponentially-large state space using reasonable memory, but that is another matter.)

The DUT is the device under test. **Blackbox testing** is where tests are phrased only in terms of the externally visible behaviour of DUT. **Whitebox testing** enables assertions to range over internal variables of the DUT.

5.1.5 Property Specification Language (PSL)

PSL is a linear-time temporal algebra designed for RTL engineering.

www.project-veripage.com/psl_tutorial_2.php

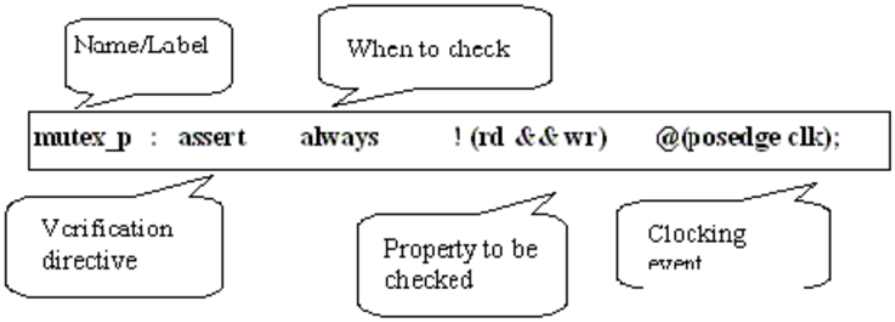


Figure 5.2: General structure of a PSL assertion

As in most temporal logics, there are three main directives:

1. always and never,
2. next (family of them),
3. eventually!

The `always` directive is the most frequently used and it specifies that the following property expression should be checked every clock. .xi The `never` directive is a shorthand for a negated `always`.

The `next` directive relates successive state properties, as qualified by the clocking event and qualifying guard.

The `eventually!` directive is for liveness properties that relate to the future. .xi The `eventually!` directive is suffixed with a bang sign to indicate it is strong property that cannot be (fully) checked with simulation.

For hands-on experience, see a previous ACS exercise: Dynamic validation using Monitors/Checkers and PSL

The general structure of a PSL assertion has the following parts:

- A name or label that can be used for diagnostic output.
- A verification directive, such as **assert**.
- When to check, such as **always** or **eventually!**.
- The property to be checked: a state expression or a temporal logic expression.
- A qualifying guard, such as a clock edge or enable signal at which time we expect the assertion to hold.

5.1.6 ABD - PSL Four-Level Syntax Structure

The abstract syntax of PSL uses for levels:

- Since the language is embedded in the concrete syntax of several other languages, such as Verilog, SystemVerilog and VHDL, its syntactic details vary. In particular, creating state predicates involves expressions that range over the nets and variables of the host language. The precise means for this is defined by the **MODELLING LAYER** that allows one to create state properties using RTL.

  Non-boolean, symbolic sub-expressions can be used in the modelling layer to generate boolean state predicates.

```
assign tempok = temperature < 99;
```

- All high-level languages and RTLs have their own syntax for boolean operators and this can be used within the modelling layer. However boolean combinations can also be formed using the PSL **BOOLEAN LAYER**.

```
not (rd and wr); // rd, wr are nets in the RTL (modelling layer).
```

- The PSL **TEMPORAL LAYER** allows one to define named sub-expressions and properties that use the temporal operators. For example:

```
// Sequence definition
sequence s1 is {pkt_sop; (not pkt_xfer_en_n [*1 to 100]); pkt_eop};

sequence s2 is {pkt_sop; (not pkt_xfer_en_n [*1 to 100]); pkt_aborted};

// Property definition
property p1 is reset_cycle_ended |=> {s1; s2};

// Property p1 uses previously defined sequences s1 and s2.
```

- The PSL **VERIFICATION LAYER** implements the declarative language itself. It includes the main keywords, such as **assert**.

.xi PSL has a rich regular expression syntax for pattern matching. These are called *SERES* or sequences. SERES stands for Sugar Extended Regular Expression, where Sugar was an older name for PSL.

Sequence elements are state properties from Modelling and Boolean layers. Core operators are (of course): disjunction, concatenation and arbitrary repetition. As a temporal logic: interpret concatenation as a time sequencing.

- **A;B** Semicolon denotes sequence concatenation
- **A[*]** Postfix asterisk for arbitrary repetition
- **A|B** Vertical bar (stile) for alternation.

Make easier to use with additional operators defined in terms of primitives:

- **A[+]** One or more occurrences: **A;A[*]**
- **A[*n]** Repeat `n` times
- **A[=n]** Repeat `n` times non-consecutively
- **A:B** Fusion concatenation (last of A occurs during first of B)

Further repetition operators denote repeat count ranges. Repeat counts must be compile-time constant (for today's standard/tools).

5.1.7 ABD - PSL Properties and Macros

PSL defines some simple path to state macros

- `rose(X)` means **!X; X**
- `fell(X)` means **X; !X**

Others are easy to define:

- `stable(X)` can be defined as **X; X || !X; !X**
- `changed(X)` can be defined as **X; !X || !X; X**

- `onehot(X)` can be defined as **X is a power of 2**
- `onehot0(X)` can be defined as **onehot(X) || (X==0)**

5.1.8 ABD - Naive Path to State Conversion

Compiling regular expressions to RTL is completely straightforward (part of a typical proof that for every RE there is an FSM).

By converting a path expression to a state expression we can generate an RTL checker for use in dynamic validation. It can also be used for converting all path expressions to state expressions if the core of a proof tool can only handle state expressions, such as a raw BDD package or SAT solver.

Additional notes:

The following ML fragment handles the main operators: concatenation, fusion concatenation, alternation, arbitrary repetition and n-times repetition.

```
fun gen_pattern_matcher g (seres_statexp e) = gen_and2(g, gen_boolean e)

|   gen_pattern_matcher g (seres_diop(diop_seres_alternation, l, r)) =
    let val l' = gen_pattern_matcher g l
        val r' = gen_pattern_matcher g r
        in gen_or2(l', r') end

|   gen_pattern_matcher g (seres_diop(diop_seres_catenation, l, r)) =
    let val l' = gen_dff(gen_pattern_matcher g l)
        val r' = gen_pattern_matcher l' r
        in r' end

|   gen_pattern_matcher g (seres_diop(diop_seres_fusion, l, r)) =
    let val l' = gen_pattern_matcher g l
        val r' = gen_pattern_matcher l' r
        in r' end

|   gen_pattern_matcher g (seres_monop(mono_arb_repetition, l)) =
        let val nn = newnet()
            val l' = gen_pattern_matcher nn l
            val r = gen_or2(l', g)
            val _ = gen_buffer(nn, r)
            in r end

|   gen_pattern_matcher g (seres_diop(diop_n_times_repetition, l,
                          seres_statexp(x_num n))) =
        let fun f (g, k) = if k=0 then g else
            gen_pattern_matcher (f(g, k-1)) l
            in f (g, n) end
```

This generates a simple one-hot automaton and there are far more efficient procedures used in practice and given in the literature.

A harder operator to compile is the length-matching conjunction (introduced shortly), since care is needed when each side contains arbitrary repetition and can declare success or failure at a number of possible times.

5.1.9 ABD - SERES Pattern Matching Example

Suppose four events are supposed to always happen in sequence:

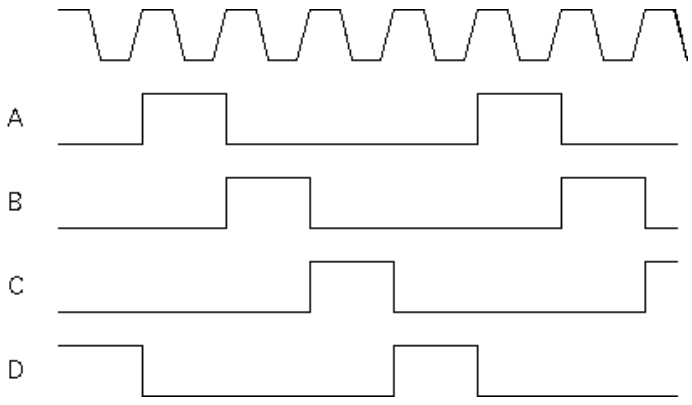
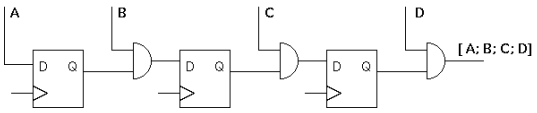


Figure 5.3: Desired behaviour of the four nets.

First attempt, we write **always true[*]; A; B; C; D** Basic pattern matcher applied to **A;B;C;D** generates:

```
  DFF(g0, A, clk);
  AND2(g1, g0, B);
  DFF(g2, g1, clk);
  AND2(g3, g2, C);
  DFF(g4, g3, clk);
  AND2(g5, g4, D);
 // Hmmm D must always hold then ?
 // Not what we wanted!
> val it = x_net "g5" : hexp_t
```



Putting a simple SERES as the body of an **always** statement normally does not have the desired effect: it does not imply that the contents occur sequentially. Owing to the overlapping occurrences interpretation, such an **always** statement distributes over sequencing and so implies every element of the sequence occurs at all times.

Therefore, it is recommended to **always uses an SERES as part of a suffix implication** or with some other temporal layer operator.

5.1.10 PSL: Further Temporal Layer Operators

.xi The disjunction (ORing) of a pair of sequences is already supported .xi by the SERES disjunction operator. .xi But PSL sequences can .xi also be combined with implication and conjunction operators in the 'temporal layer'.

- `P |-> Q` P is followed by Q (one state overlapping),
- `P |=> Q` P is followed by Q (immediately afterwards),
- `P && Q` P and Q occur at once (length matching),
- `P & Q` P and Q succeed at once,
- `P within Q` P occurred at some point during Q,
- `P until Q` P held at all times until Q started,
- `P before Q` P held before Q held.

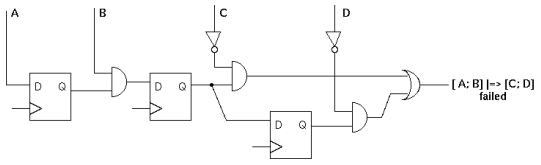
5.1.11 ABD - Sequence Constraint as a Suffix Implication

Earlier example: add a **onehot** assertion - that will constrain the state space. Also, consider some phrasing using suffix implications to constrain the state trajectory:

```
// (Verilog concatenation braces, not a PSL sequence).
always onehot ({A,B,C,D});
// expands to
```

```
>val it = // holds on error
  (((A<<3)|(B<<2)|(C<<1)|D) != 8) &&
  (((A<<3)|(B<<2)|(C<<1)|D) != 4) &&
  (((A<<3)|(B<<2)|(C<<1)|D) != 2) &&
  (((A<<3)|(B<<2)|(C<<1)|D) != 1);

//(ML for expanding above macro not in notes)
```

```
// A feasible-looking suffix implication:

always { A;B } |=> { C;D };

// It expands to:
```




```
  DFF(g0, A, clk);
  AND2(g1, g0, B);
  DFF(g2, g1, clk);
  INV(g3, C);
  AND2(g4, g3, g2);  // Holds if C missing
  DFF(g5, g2, clk);
  INV(g6, D);
  AND2(g7, g5, g6);  // Holds if D missing
  OR2(g8, g7, g4);
> val it = x_net "g8" : hexp_t   // Holds on error
```

Even this is not very specific: C and D might occur at other times. It is a good idea to write protocol rules as suffix implications that range over SERES. Use a separate temporal implication for each sequential step.

What about asserting a requirement of **data conservation** ? At an interface we commonly want to assert that data is not lost or duplicated. Is PSL any help? Not really, one needs a language that can range over symbolic data and tagged streams of data.

5.1.12 Automated Stimulus Generation (Directed-Random Verification)

Commerical products: Verisity's Specman Elite www.open-vera.com

Simulations and test programs require **stimulus**. This is a sequence of input signals, including clock and reset, that exercise the design.

Given that formal specifications for many of the input port protocols might exist, one can consider automatic generation of the stimulus, from random sources, within the envelope defined by the formal specification. Several commercial products do this, including Verisity's Specman Elite, Synopsys Vera.

Here is an example of some code in Specman's own language, called 'e', that defines a frame format used in networking. Testing will be inside envelope defined by **keep** statement.

```
struct frame {
  llc: LLCHeader;
  destAddr: uint (bits:48);
  srcAddr: uint (bits:48);
  size: int;
  payload: list of byte;
  keep payload.size() in [0..size];
};
```

Sequences of bits that conform to the frame structure are created and presented at an input port of the design under test. An heirarchy of specifications and constraints is supported. One can compose and extend one specification to reduce its possible behaviours:

```
  // Subclass the frame to make it more specialised:
  extend frame { keep size == 0;  };
```

5.1.13 OVM/UVM

The **Open Verification Methodology** (OVM) is a documented methodology with a supporting building-block library for the verification of semiconductor chip designs.

Doulos: From OVM to UVM

run OVM simulations from a web browser

Verification Methodology Cookbooks

5.1.14 ABD - A Simple Model Checker

For a small finite state machine we can use a simple model checker for a state safety property:

Algorithm: **'Find reachable state space'** (add successors of current set until closure):

1. S := { q_0 } // initial state
2. S := S $\cup$ $\{q' \mid \exists\ \sigma \in \Sigma, q \in S\ .\ NSF(q,\sigma) = q'\ \}$
3. If safety property does not hold in any $q \in S$ then flag error.
4. If S increased in step 2 then goto step 2.

S can be held explicitly in bit map form or symbolically as a BDD.

Variation 1: ignore safety property while finding reachable state space then finally check for all found states.

Variation 2: property to check might be a path property, so either

- Compile it to a checking automaton (becomes a state property of expanded NSF), or
- Expand it as we go (using modal mu calculus).

The PSL strong assertions need to be checked with a formal proof tool. Model checking is normally used because it is fully automated.

A model checker explores every possible execution route of a finite-state system by exploring the behaviour over all possible input patterns.

There are two major classes of model checker: explicit state and symbolic. Explicit state checkers actually visit every possible state and store the history in a very concise bit array. If the bit array becomes too big they use probabilistic and hashing techniques. The main example is Spin. Symbolic model checkers manipulate expressions that describe the reachable state space and these were famously implemented as BDDs in the SMV checker. There are also other techniques, such as bounded model checking, but the internal details of model checkers is beyond the scope of this course.

The most basic model checker only checks state properties. To check a path property it can be compiled into an automaton and included as part of the system itself.

To check liveness formally is beyond the scope of this course, but one algorithm is to repeatedly trim cul-de-sacs from the state transition graph so that only a core where all states are reachable from all others remains.

5.1.15 ABD - Boolean Equivalence Checker

Boolean equivalence: do the two functions produce the same output?

- For all input combinations ?
- For a subset of input combinations (some input patterns are don't cares).

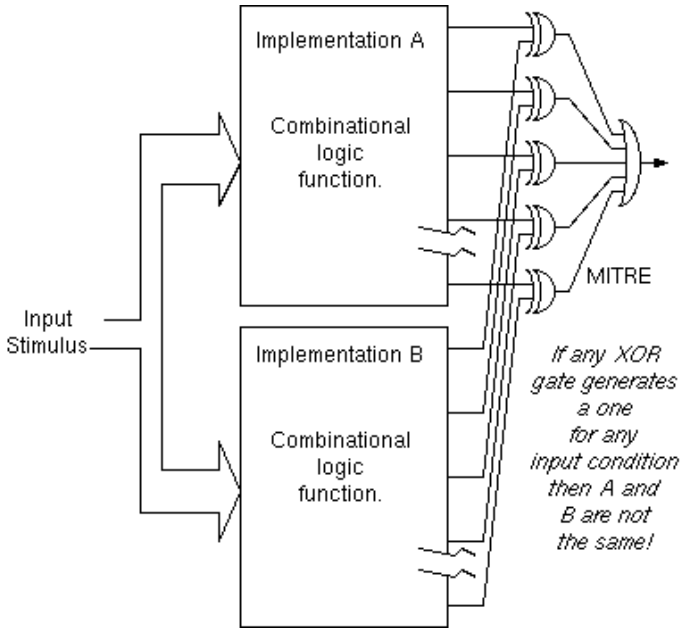


Figure 5.4: A mitre compares the outputs from a pair of supposedly-equivalent combinational components.

Often we have two implementations to check for equivalence, for instance, when RTL is turned into a gate-level netlist by synthesis we have:

- RTL version: pre-synthesis, and
- Gate-level version: post-synthesis.

Sources of difference between the designs might be manual implementation of one of them, manual edits to synthesiser outputs and EDA tool faults. For instance, after place and route operations, it is common to extract the netlist out from the masks and check that for correctness, so this is another source of the same netlist.

The **boolean equivalence problem** is do two functions produce the same output. However, are we interested for all input combinations? No, normally we are only interested in a subset of input combinations (because of don't care conditions).

The method, shown in Figure 5.4, is to create a **mitre** of the two designs using a disjunction of XOR gate outputs. Then, feed negation of mitre to a SAT solver to see if it can find any input condition that produces a one on the output.

SAT solving is a matter of trying all input combinations, so has exponential cost in theory and is NP complete. However, modern solvers such as **zChaff** essentially exploit the intrinsic structure of the problem so that they normally are quite quick at finding the answer.

Result: if there are no input combinations that make the mitre indicate a functionality difference, then the designs are equivalent.

Commercial example: Synopsys Formality

5.1.16 ABD - Sequential Logic Equivalence

The figure shows implementations of a two-bit shift register. They differ in amount of internal state. They have equivalent observable behaviour (ignoring glitches). Note, to implement larger delays, the design based on

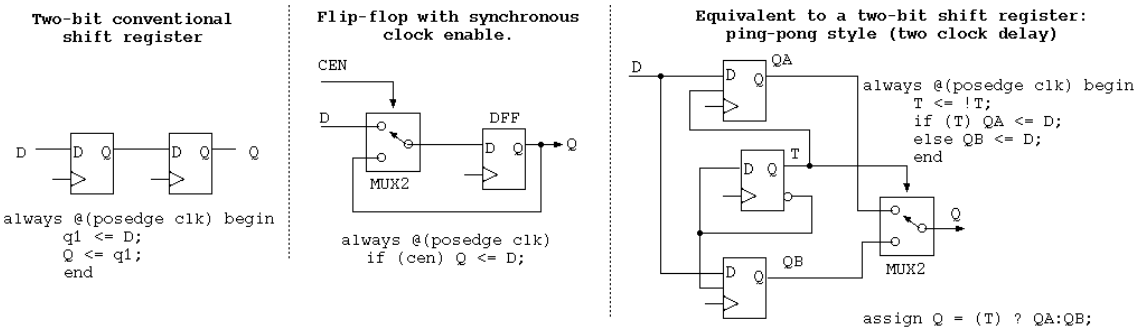


Figure 5.5: Two circuits that use different amounts of internal state to achieve the same functionality.

multiplexors might use more logic and less power then the design based on shifting, since fewer nets toggle on each clock edge.

Another common question that needs checking is *sequential equivalence*. Do a pair of designs follow the same state trajectory ?

- Considering the values of all state variables ?
- Considering a re-encoding of the state variables ?
- For an observable subset of the state (e.g. at an interface) ?
- When interfacing with a given reactive automaton ?

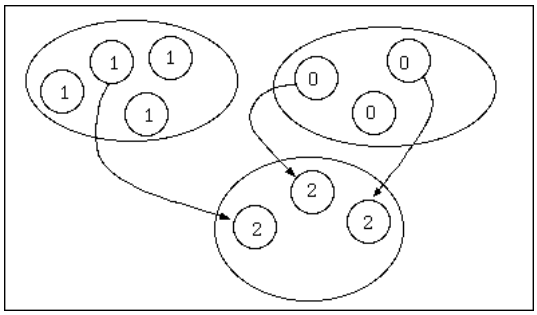
Other freedoms that could be allowed within the notion of equivalence:

- Temporally floating ports - latency independence. .xi With floating ports we do not consider the relative timing of events between ports, only the relative timing of events within each port.
- Synchronous or asynchronous (turn-taking) composition. .xi If a pair of circuits are combined, do they share a common clock or take it in turns to move?
- Strong or weak bi-simulation (stuttering equivalence). .xi A stuttering equivalence between a pair of designs may exist if we disregard the precise number of clock cycles each took to achieve the result (such as different implementations of a microprocessor).

Practical problem: Designs may only be equivalent in the used portion of the state space. .xi Hence we may need a number of side conditions that specifiy the required operating conditions.

5.1.17 ABD - Sequential Logic Simplification

A finite-state machine may have more states than it needs to perform its observable function because some states are totally equivalent to others in terms of output function and subsequent behaviour. Note that one-hot coding does not increase the reachable state space and so is **not** an example of that sort of redundancy.



A Moore machine can be simplified by the following procedure:

- 1. Partition all of the state space into blocks of states where the observable outputs are the same for all members of a block.
- 2. Repeat until nothing changes (i.e. until it closes) For each input setting:
  - 2a. Chose two blocks, B1 and B2.
  - 2b. Split B1 into two blocks consisting of those states with and without a transition from B2.
  - 2c. Discard any empty blocks.
- 3. The final blocks are the new states.

Bisimulation algorithm not examinable in this course.

Alternative algorithm: start with one partition per state and repeatedly conglomerate. The best algorithms use a mixture of the two approaches to meet in the middle. Wikipedia: Formal Equivalence Checking

Research example: CADP package: developed by the VASY team at INRIA. Commercial products: Conformal by Cadence, Formality by Synopsys, SLEC by Calypto.

One future use of this sort of procedure might be to generate an instruction set simulator for a processor from its full RTL implementation. This sort of de-pipelining would give a non-cycle accurate, higher-level model that runs much faster in simulation.

There are some good on-line resources. Such as Dulos System Verilog Assertions

5.1.18 ABD - Conclusion

ABD today is often focussed on safety and liveness properties of systems and formal specifications of the protocols at the ports of a system. However, there are many other useful properties we might want to ensure or reason about, such as those involving counting and/or data conservation. These are less-well embodied in contemporary tools.

PSL deals with concrete values rather than symbolic values. Many interesting properties relate to symbolic data (e.g. specifying the correct behaviour of a FIFO buffer). Using PSL, all symbolic tokens must be wrapped up in the modelling layer which is not the core language.

Formal methods are taking over from simulation, with the percentage of bugs being found by formal methods growing. However, there is a lack of formal design entry. Low-level languages such as Verilog do not seamlessly mix with automatic synthesis from formal specification and so double-entry of designs is common.

SG 6 — SystemC: Hardware Modelling Library

Topics: SystemC.

6.1 SystemC: Hardware Modelling Library for C++

SystemC is a free library for C++ for hardware SoC modelling. Download from www.accelera.org SystemC was developed over the last ten years. There have been two major releases, 1.0 and 2.0. Also of importance is the TLM sub-library, TLM 2.0. (SystemC using transactional-level modelling (TLM/ESL) is covered later).

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

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

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

SystemC includes (at least):

- A module system with inter-module channels: C++ class instances are instantiated in a hierarchy, following the circuit component structure, in the same way that RTL modules instantiate each other.
- An eventing and threading kernel that is non-preemptive and which allows user code inside components to run either in a trampoline style, returning the thread without blocking, or to keep the thread and hold state on a stack.
- Compute/commit signals as well as other forms of channel for connecting components together. The compute/commit signals are needed in a zero-delay model of hardware to avoid 'shoot-thru': i.e. the scenario where one flip-flop in a clock domain changes its output before another has processed the previous value.
- A library of fixed-precision integers. Hardware typically uses all sorts of different width busses and counters that wrap accordingly. SystemC provides classes of signed and unsigned variables of any width that behave in the same way. For instance the user can define an `sc_int` of five bits and put it inside a signal. The provided library includes overloads of all the standard arithmetic and logic operators to operate on these types.
- Plotting output functions that enable waveforms to be captured to a file and viewed with a program such as `gtkwave`.
- A transactional modelling sub-library: TLM 1.0 provided separate blocking and non-blocking interfaces prototypes that a user could follow and in TLM 2.0 these are rolled together into 'convenience sockets' that can convert between the two forms.

Problem: hardware engineers are not C++ experts but they can be faced with complex or advanced C++ error messages when they misuse the library.

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

```
SC_MODULE(mycounter) // An example of a leaf module (no subcomponents).
{
   sc_in  < bool       > clk, reset;
   sc_out < sc_int<10> > myout;

   void m()  // Internal behaviour, invoked as an SC_METHOD.
   {
      myout =  (reset) ? 0: (myout.read()+1); // Use .read() since sc_out makes a signal.
   }

   SC_CTOR(mycounter)  // Constructor
     { SC_METHOD(m);   //
       sensitive << clk.pos();
     }
}
// Complete example is on course web site and also on PWF.
```

SystemC enables a user class to be defined using the the SC_MODULE macro. Modules inherit various attributes appropriate for an hierarchic hardware design including an instance name, a type name and channel binding capability. The `sensitive` construct registers a callback with the EDS kernel that says when the code inside the module should be run. An unattractive feature of SystemC is the need to use the .read() method when reading a signal.

6.1.1 SystemC Abstracted Data Modelling

Here we raise the modelling abstraction level by passing an abstract datatype along a channel. the abstract data type must define a few basic methods, such as the equality operator overload this is shown:

```
  sc_signal < bool > mywire; // Rather than a channel conveying just one bit,

  struct capsule
  { int ts_int1, ts_int2;
    bool operator== (struct ts other)
    { return (ts_int1 == other.ts_int1) && (ts_int2 == other.ts_int2); }

    int next_ts_int1, next_ts_int2; // Pending updates
    void update()
    { ts_int1 = next_ts_int1; ts_int2 = next_ts_int2;
    }

    ...
    ...  // Also must define read(), write() and value_changed()


  };

  sc_signal < struct capsule > myast; // We can send two integers at once.
```

For many basic types, such as `bool, int, sc_int,` the required methods are provided in the SystemC library, but clearly not for user-defined types.

```
  void mymethod()  { .... }
  SC_METHOD(mymethod)
  sensitive << myast.pos(); // User must define concept of posedge for his own abstract type.
```

6.1.2 Threads and Methods

SystemC enables a user module to keep a thread and a stack but prefers, for efficiency reasons if user code runs on its own upcalls in a trampoline style.

- An `SC_THREAD` has a stack and is allowed to block.

- An `SC_METHOD` is just an upcall from the event kernel and must not block.

Comparing SC_THREADs with trampoline-style methods we can see the basis for two main programming TLM styles to be introduced later: blocking and non-blocking.

The user code in an SC_MODULE is run either as an SC_THREAD or an SC_METHOD.

An SC_THREAD has a stack and is allowed to block. An SC_METHOD is just an upcall from the event kernel and must not block. Use SC_METHOD wherever possible, for efficiency. Use SC_THREAD where important state must be retained in the program counter from one activation to the next or when asynchronous active behaviour is needed.

The earlier 'mycounter' example used an SC_METHOD. Now an example using an SC_THREAD: a data source that provides numbers using a net-level four-phase handshake:

```
SC_MODULE(mydata_generator)
{ sc_out < int  > data;
  sc_out < bool > req;
  sc_in  < bool > ack;

  void myloop()
  { while(1)
    { data = data.read() + 1;
      wait(10, SC_NS);
      req = 1;
      do { wait(10, SC_NS); } while(!ack.read());
      req = 0;
      do { wait(10, SC_NS); } while(ack.read());
    }
  }

  SC_CTOR(mydata_generator)
  {
    SC_THREAD(myloop);
  }
}
```

A SystemC thread can block for a given amount of time using the `wait` function in the SystemC library (not the Posix namesake). NB: If you put '`wait(4)`' for example, you will invoke the unix system call of that name, so make sure you supply a SystemC time unit as the second argument.

Additional notes:

Waiting for an arbitrary boolean expression to hold is hard to implement on top of C++ owing to its compiled nature:

- C++ does not have a reflection API that enables a user's expression to be re-evaluated by the event kernel.
- Yet we still want a reasonably neat and efficient way of passing an uninterpreted function.
- Original solution: the delayed evaluation class:

```
    waituntil(mycount.delayed() > 5 && !reset.delayed());
```

Poor user had to just insert the **delayed** keyword where needed and then ignore it when reading the code. It was too unwieldly, now removed. So today (pre C++11) use the less-efficient:

```
    do { wait(10, SC_NS); }  while(!(mycount > 5 && !reset)));
```

6.1.3 SystemC Plotting and GUI

We can plot to industry standard VCD files and view with gtkwave (or modelsim).

```
sc_trace_file *tf = sc_create_vcd_trace_file("tracefile");

// Now call:
// sc_trace(tf, <traced variable>, <string>);

sc_signal < int > foo;
float bar;
sc_trace(tf, foo);
sc_trace(tf, bar, "bar"); // Give name if anon constuctor

sc_start(1000, SC_NS);  // Simulate for one microsecond
sc_close_vcd_trace_file(tr);
return 0;
```

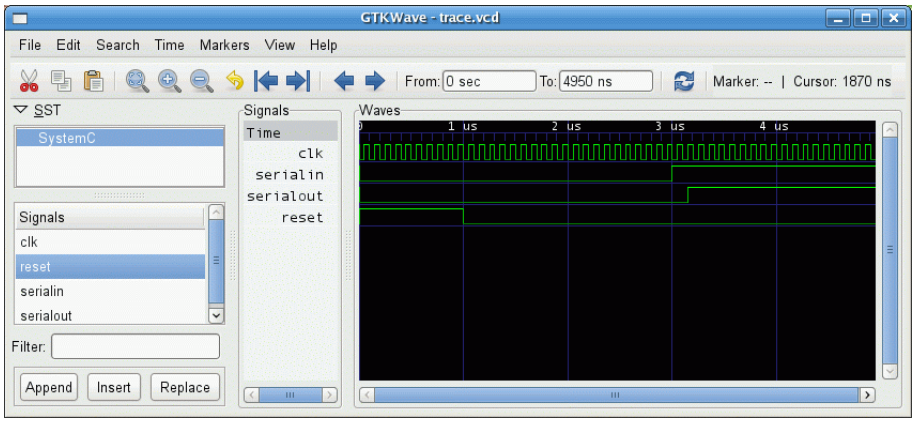


Figure 6.1: Waveform view plotted by gtkwave.

VCD can be viewed with **gtkwave** or in **modelsim**. There are various other commercial interactive viewer tools...

Try-it-yourself on PWF

SG 7 — ESL: Electronic System Level Modelling

Recall the following levels of modelling from the start of this course:

- **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.

Chosen baseline methodolody: SystemC Transactional Modelling using high-level models in C++. Enhancements:

- Synthesise high-level models to form parts of the fabricated system (see elsewhere notes on 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 5).

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 sub-class 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.

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

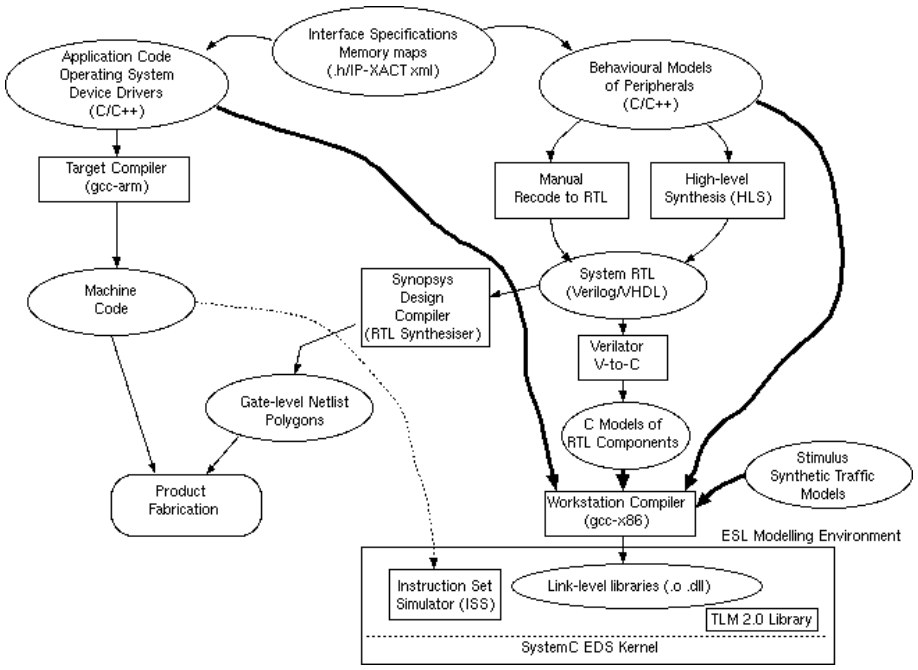


Figure 7.1: ESL Flow: Avoiding the ISS by cross-compiling the firmware and direct linking with behavioural models.

Our ESL flow is mainly 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 the target processor machine code.

7.1.1 Using C Preprocessor to Adapt Firmware

We may need to recompile the hardware/software interface when compiling for TLM 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.
```

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

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

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

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

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

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

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

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

A full example is in the 'ethercrc.zip' folder on the course web site (and unzipped on PWF).

Alternatively, it is also possible to use the workstation VM system to trap calls from natively-compiled firmware to hardware.

7.2 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 net or bus,
2. **Abstract data modelling**: an event is delivery of a complete cache line or other data packet,
3. **Transactional-level modelling**: avoid events as much as possible: use intermodule software calling.

In general, 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 relatively intact.

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.

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.

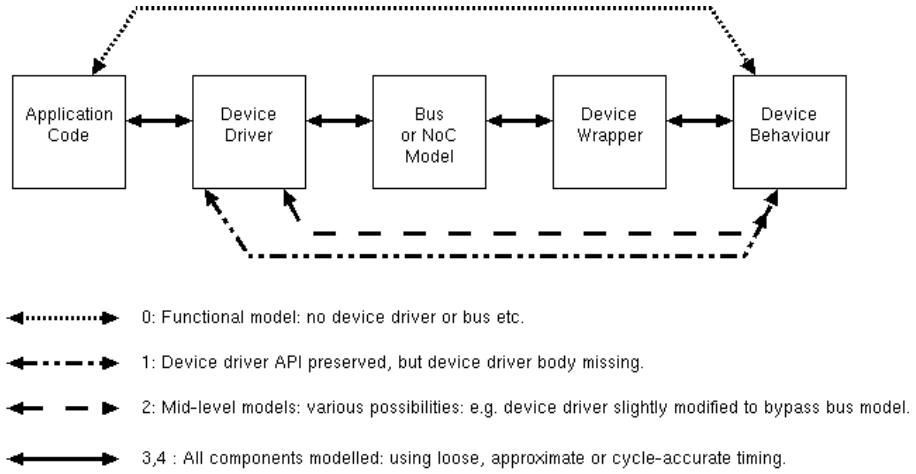


Figure 7.2: Some possible short-cuts through full system model to omit details.

Consider the Ethernet CRC example

Another useful taxonomy over the higher modelling abstractions:

1. Highest-level (vanished) model: 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. Low-level model: Implemented in RTL or cycle-accurate SystemC: target device driver firmware and other code is used unmodifed.

Figure 7.3 is an example protocol implemented at net-level and TLM level:

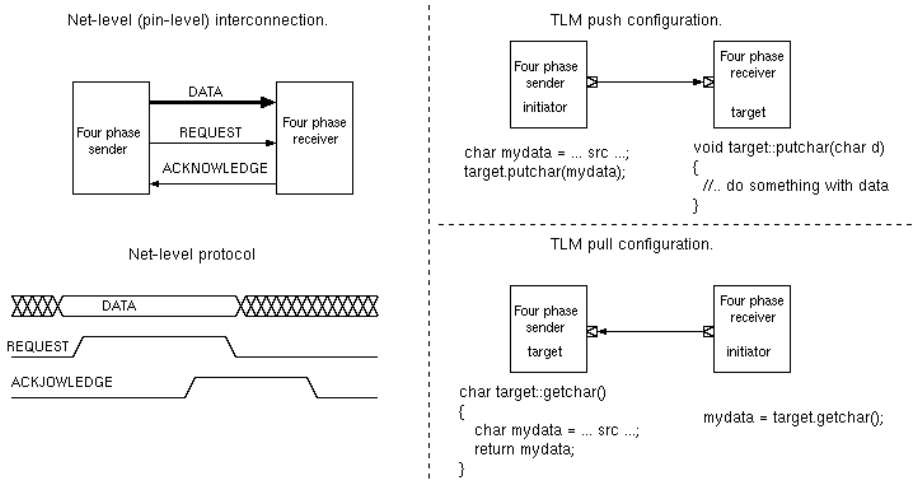


Figure 7.3: Three views of four-phase handshake between sender and receiver: net-level connection and TLM push and TLM pull configurations (untimed).

Note that the roles of initiator and target do not necessarily relate to the sources and sinks of the data. In fact, an initiator can commonly make both a read and a write transaction on a given target and so the direction of data transfer is dynamic.

7.2.1 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.

Example **blocking transactors**: convert from transaction to pin-level modelling.

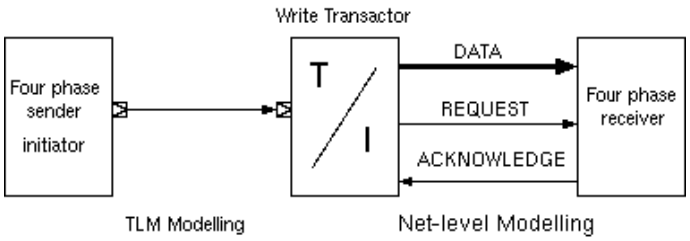


Figure 7.4: Mixing modelling styles using a transactor.

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

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

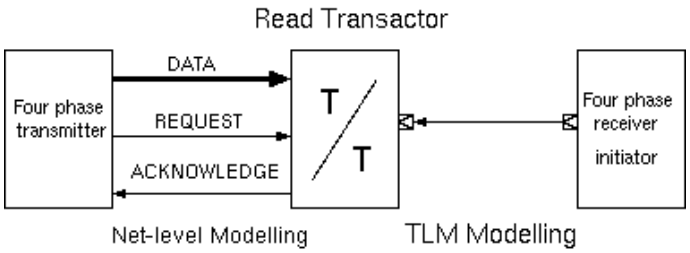


Figure 7.5: Mixing modelling styles using a transactor 2.

'Toy ESL'practical material.

7.2.2 Transactor Configurations

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

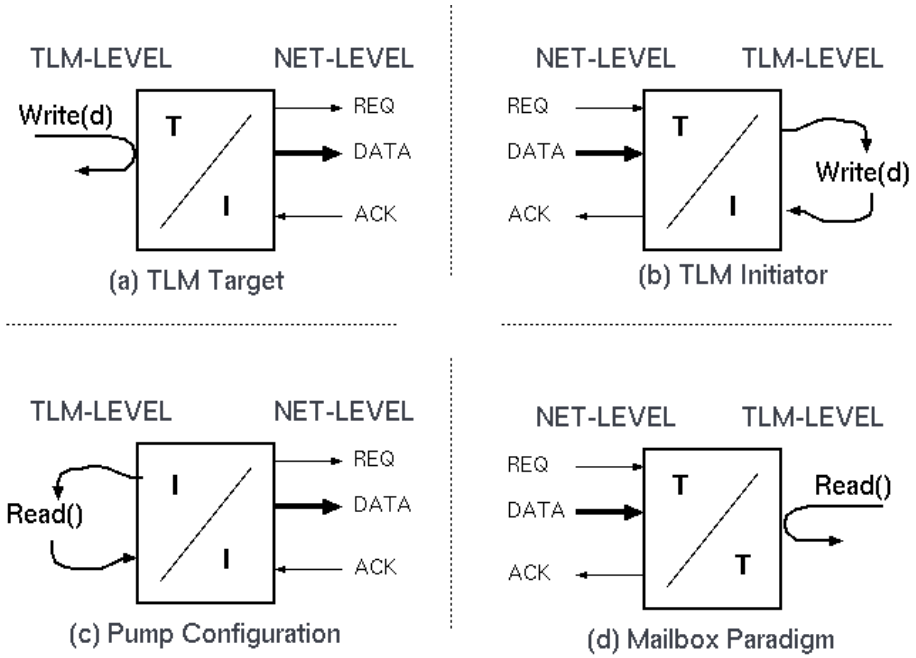


Figure 7.6: Possible configurations for simple transactors.

Additional notes:

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

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

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

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

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

7.2.3 ESL TLM in SystemC: First Standard TLM 1.0.

NB: Full exam 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.

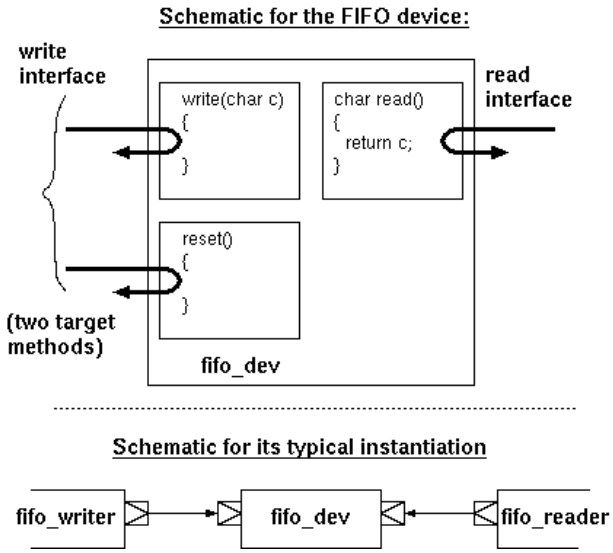
However, TLM 1.0 had no standardised or recommended structure for payloads and no standardised timing annotation mechanisms.

There was also the problem of how to have multiple TLM ports on a component with same interface: e.g. a packet router.

However, referring back to the DMA unit behavioural model, 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):

```
//Define the interfaces:
class write_if: public sc_interface
{ public:
  virtual void write(char) = 0;
  virtual void reset() = 0;
};

class read_if: public sc_interface
{ public:
  virtual char read() = 0;
};

//Define a component that inherits:
class fifo_dev: sc_module("fifo_dev"),
public write_if, public read_if, ...
{
  void write(char) { ... }
  void reset() { ... }

  ...
}
```




```
SC_MODULE("fifo_writer")
{
  sc_port< write_if > outputport;
  sc_in < bool > clk;
  void writer()
  {
    outputport.write(random());
  }

  SC_CTOR(fifo_writer} {
     SC_METHOD(writer);
     sensitive << clk.pos();
  }
}

//Top level instances:
fifo_dev myfifo("myfifo");
fifo_writer mywriter("mywriter");
// Port binding:
mywriter.outputport(myfifo);
```

Here a thread passes between modules, but modules are plumbed in Hardware/EDS netlist structural 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.

However we soon run in to the well-known OO problem with multiple instances of an interface: not often needed for S/W but common enough in H/W designs.

7.2.4 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 reference time stamp, but we aim not to use the kernel very much, only entering it when inter-module communication is needed. 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.

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

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

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

.xi 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:
sc_wait(delay);
delay = 0;
```

Important note: **Simulation performance is reduced when there are frequent resynchs, but true transaction ordering will be modelled correctly.**

7.2.5 Instruction Set Simulator (ISS)

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

```
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):

```
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;
    }
}
```

'Toy ESL'practical material.

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 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 (as described later) exposes the unfettered raw performance of the workstation for cpu-intensive code.

7.2.6 Typical ISS setup with Loose Timing (Temporal Decoupling)

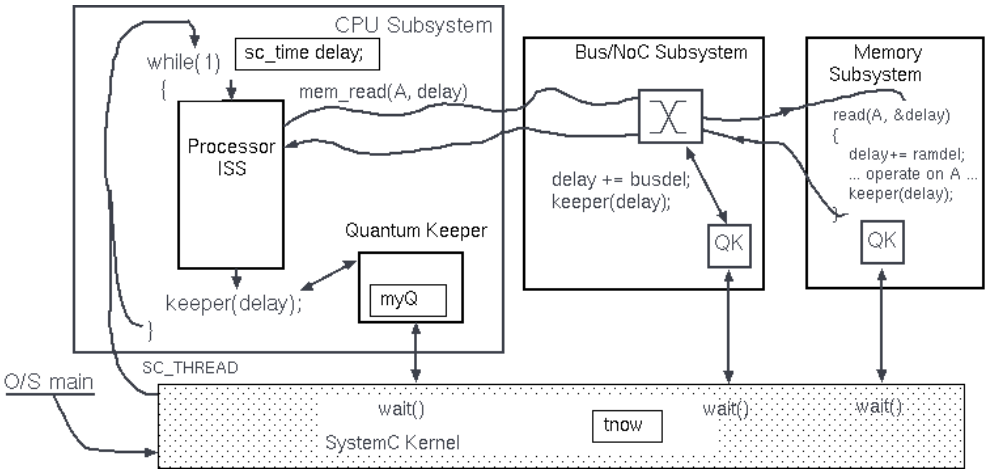


Figure 7.7: Typical setup of thread using loosely-timed modelling with a quantum keeper.

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. 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.

Each component may have a quantum keeper. Every thread must encounter a quantum keeper at least once in its outermost loop.

The quantum keeper code is just a conditional resynch:

```
if (delay > myQ) { sc_wait(delay); delay = 0; }
```

By calling `wait(delay)` the simulation time will advance to where the caller has got to while running other pending processes. .xi The myQuantum could be a system default value or a special value for each thread or component.

Or where a thread needs to block to wait for a result from some other thread:

```
while (!condition_of_interest)
{
  sc_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 ?

7.2.7 RTL Power Estimation Without Simulation

Post RTL synthesis we have a netlist and can use Rent for wire lengths provided sufficent hierachy 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)

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

```
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:

- Flip-flop activity (number of enabled cycles/number of flipping bits)
- Number of writes to RAMs
- Glitch energy in combinational logic.

7.2.8 Typical macroscopic performance equations: SRAM example.

It is important to model SRAM accurately. A 45nm SRAM can be modelled simply in terms of 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 = 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).

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

Activity ratio, a: is the percentage of clock cycles that see a transition. The net toggle rate = Operating frequency of the chip $f \times a$;

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

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

7.2.9 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.

7.2.10 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 our net-level SystemC simulations.

SAIF: Switching Activity Interchange Format - the industry standard approach (aka Spatial Archive Interchange Format). Quick Tutorial and also TKT 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.

7.2.11 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 organization 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 gives a simple power-law relationship for the number of wires to a logic block. Wire length distribution (with good placement) follows an equally-predictable pattern. With a heirarchic design, where we have the area use of each leaf cell, even without placement, we can follow a net's trajectorary 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.

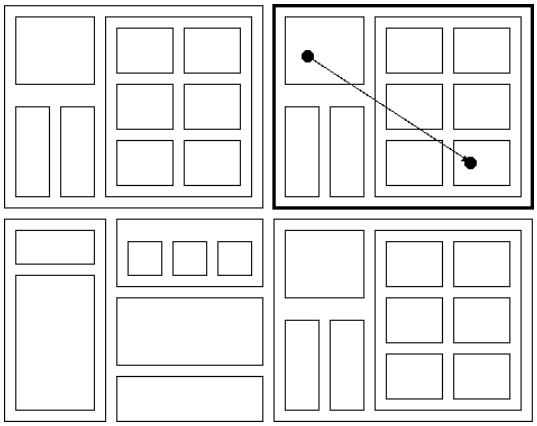


Figure 7.8: Illustrating Lowest Common Parent of the endpoint logic blocks. (This will always be roughly the same size for any sensible layout, so having a detailed layout like the one shown is not required).

7.2.12 Macroscopic Phase/Mode Power Estimation Formula

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 END.