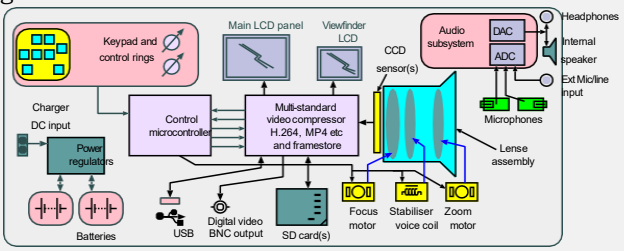
Modern SoC Design on Arm
End-of-Chapter Exercises

Q1 Product Design

Consider the design of a high-quality digital movie camera. Sketch a feasible top-level block diagram, remembering to include a viewfinder and audio subsystem, but you may ignore auto-focusing. How many circuit boards, SoCs and processors should it have? To mitigate against the camera shaking, what are the relative costs of implementing a vision stabilizer using voice-coil prism hardware compared with an electronic/software-only implementation.

Answer:

For students without familiarity with top-end video cameras, the first step would be to download a relevant user manual or product flier from Panasonic, Sony or Canon and perhaps search for relevant papers, such as *Integration of image stabilizer with video codec for digital video cameras*, YC Peng et. al. at ISCAS05.



General internal block diagram of a professional video camera. A lot of further components could be present, such as WiFi, RTC and ND filters.

The first design consideration is to estimate how much data needs to flow along what paths. There are essentially three forms of data:

video: Let us assume UHD video format: 3840×2160 at 30 fps (for USA) at 24 bits per pixel raw picture format. This is 746 MByte/second (MB/s). After compression these target class 4 SDXC memory cards at 25 MB/s.

audio: Stereo audio at 24 bits, 48 ksps uses 288 KB/s.

control: On-screen time code and status display and prism control for antishake (and auto-focussing) might require bandwidths of up to a 1 KB/second.

Clearly the control data can be managed using programmed I/O on a 'host' central controlling processor or micro-controller that manages the push buttons, keypads and all ancillary functions, from battery charge monitoring and turning the unit on and off. Having a dedicated processor for all of this is probably a good idea, since it can be always powered up, but maintained in an ultra-low power sleep mode while the user thinks the device is switched off. It will

have a lot of low-rate connections to all the other hardware sub-systems and to all the buttons. It might be best situated on the reverse of a membrane switch control panel PCB. If a USB webcam facility is provided via this processor, it would be perhaps low rate only. Main live digital video output in a studio use case would likely be via a dedicated socket (e.g. SDI on BNC socket).

The audio subsystem needs an electrically quiet environment to meet professional standards. Pro-audio signal-to-noise ratios need to be better than 100 dB. This can easily be achieved by putting the microphone pre-amplifiers and ADCs on a separate PCB and this might also suit camera ergonomics where the microphone and audio input sockets tend to be on the top front of the camera. DACs for replay and headphone/speaker amplifiers might be placed on the same circuit board. A simple, synchronous digital serial interface to the main video hardware would likely be used.

The main video capture and compression assembly is likely to be built on one main PCB behind the lens, with CCD sensors connected to an ASIC that implements compression. For MPEG compression, just two or three frames need to be stored for motion interpolation. Using the figures quoted above, one frame is 25 MByte, so this will fit in on-chip SRAM and DRAM may not be needed if compression is real time. There will also be a video output to a flat screen display, viewfinder display and HDMI output etc. The SD card or cards might be on another PCB, or perhaps could be accessed out of the side of the main PCB.

As shown in the sketch, the control processor would write on-screen display information to a framestore inside the video system.

Video stabilisation uses two components in a negative feedback loop: motion detection and adjustment of the captured image. Motion detection in software requires motion comparison between one frame and another. This is highly CPU-intensive. However, this is required as part of MPEG compression and so is already being performed if the video is being saved as MPEG in real-time as mentioned above. [Real-time MPEG compression is not always used however.] Motion detection can also be achieved using a cheap solid-state MEMS gyroscope (the key component in a modern drone). As mentioned in the question, motion correction of the captured image can be implemented using a moving prism or lens, typically mounted in a 'voice coil' actuator, so called because it works the same way as a loudspeaker (a lightweight induction coil in a permanent magnetic field). Such arrangements are cheap and mature and were a key technology in the development of spinning optical disks, like the CD. Some mobile phone cameras today have them. Alternatively, motion correction can be performed in software provided the captured image is larger than needed to allow a dynamic margin to be cropped as part of the compression chain. A top-quality video camera will likely incorporate all of these techniques.

Q2 Memory Bandwidth

An algorithm performs a task that is essentially the same as completing a jigsaw puzzle. Input values and output results are to be held in DRAM. Describe input and output data formats that might be suitable for a jigsaw with a plain image but no mating edge that can falsely mate with the wrong edge. The input data set is approximately 1 Gbyte. By considering how many DRAM row activations and data transfers

are needed, estimate how fast this problem can be solved by a uniprocessor, a multi-core PRAM model and a hardware accelerator. State any assumptions.

Answer:

Note this exercise is deliberately vague and abstract to encourage the student to take many steps back and to think about general principles from algorithmics.

Classes of jigsaw-like algorithms are widely studied in academic literature — they are relevant for genome assembly. The two essential stages are identifying which edges can bond with which others and then using this information to form the complete picture. Either phase could dominate the problem. Although we have approximately 1 GB of input data, it is not made clear how much this will be reduced by the first phase as input to the second phase. In the absence of further information, we could assume a factor of 2 to 1.

If more details are available, a simple means to get an accurate answer is to write a program in C++ and run it on a virtual platform where all memory operations are instrumented, such as the Prazor/VHLS platform used in this book. However, an approximate analytic model is another approach that is useful, either in addition or standalone. This will give the number of DRAM activations.

The first stage could be dominated by algorithmic complexity but is likely to process each piece once, so will have linear cost. The second stage is a look-up/matching process and is likely to have cost $n \log(n)$ or quadratic at worst. The second stage is highly likely to be limited by data memory bandwidth since the output of the first stage will essentially be a large, probably sparse, symmetric Boolean array.

We can assume an efficient implementation. It will not make any difference whether a hardware accelerator is used or an efficient implementation in C++ on a general purpose processor: the memory footprint will be the same if it is the same algorithm (or indeed same code compiled by HLS). Instruction fetch overhead for a software implementation can be ignored since the small amount of code required in the inner loops will all fit in a typical I-cache.

A jigsaw solving approach when there is no picture (termed *colourless* in jigsaw combinatronics) has less to go on than the standard human approach, but identifying the edge pieces and constructing an outer frame might still be a good starting point. However, for large problems, the amount of edge is vanishingly small and, as the problem scales up, this part of the problem will not significantly affect the run time.

As said, the problem likely has two phases: working out which faces of the pieces can possibly mate which others and assembling the main result. We are not told how easy the first step is, but since we are free to design the input format, we can decide that the mating condition is explicit. Hence the input data is a list of pieces that we assume are four-sided where each side is labelled with a joint number that is either zero for an outer edge or must match the joint number of its mate. Hence there is no cost to the first stage, but the data might need to be transposed from piece-centric to edge-centric. The output would be the (x,y) coordinate of each piece along with its rotation, which is one of four.

A naive approach to the second phase would be brute-force recursive search over all compatible prefixes in the full set of permutations. If we assume each piece has four sides and is represented as quadruple of the four edge numbers it has, there would be $10^9/32 = 31 \times 10^6$ pieces.

Following a raster scan placement, two sides (left and above) will be constrained to match against already placed pieces, but each piece can be placed four ways around. So, a very important factor is the number of distinct edges, since if this is small there will be an infeasibly large number of permutations, but as it grows, the number of placements decreases quadratic (since two edges are constrained). For instance, if there are 100 edge types, one in 10^4 of the available pieces can be tried next, giving perhaps 3100! possible combinations.

From a quick online literature search relating to solving colourless jigsaws (and coloured ones which arise from paper shredders!) we can see the complexity is indeed alarming. Even clever algorithms struggle. '*Solving Jigsaw Puzzles By The Graph Connection Laplacian*' by Huroyan in 2022 needed 3000 seconds to solve a $28 \times 28 = 787$ piece jigsaw on a 2.3 GHz Macbook.

In general, the number of DRAM activations will be proportional to the number of non-adjacent memory lookups. The linear scan through the pieces, pre-processing them, is likely to read each piece once, so the read activations will be 1 GByte divided by the row size, which is typically 1024 bits, hence $10^9 * 8/1024 = 7.8 \times 10^6$ and a similar computation applies for the write-back of the preprocessed data. For the second stage, we can assume the relatively small data structure for the assembled jigsaw is held in cache or on-chip SRAM for an accelerator solution, so assuming the brute-force approach, a linear scan of the preprocessed data will serve to find each next candidate. Hence, excessive amounts of random access to DRAM are avoided in both phrases. On the other hand, more advanced algorithms may vary!

A PRAM model, where a number of processing elements operate on the same DRAM might not be a good idea for the second phase. The preprocessed tile data is read-only and is probably better placed in different banks (i.e. DRAM channels) for increased read bandwidth. This is not strictly a PRAM model (although it could be supported in a flat address space, PRAM, NUMA design, with remote access being rare). A genuine PRAM approach where different processing elements start from different corners (e.g., four corners for a literal 2D jigsaw) is also a good design approach and could give a factor of 4 speed-up assuming the have-likely-met-in-the- middle heuristic is lightweight.

Q3 Parallel Processing

Two processes that run largely independently occasionally have to access a stateless function that is best implemented using about 1 mm^2 of silicon. Two instances could be put down or one instance could be shared. What considerations affect whether sharing or replication is best? If shared, what sharing mechanisms might be appropriate and what would they look like at the hardware level?

Answer:

The decision to share or keep separate is influenced by several aspects.

Traditionally, silicon area was an important consideration, so if the utilisation of the FU was

low, having a shared instance would be good from an area point of view. This is especially so if the FU is stateless, since no per-customer context bank needs to be kept. The contention arising from queuing delay can be estimated using simple queuing models. For instance, its service time might be deterministic (which means 'constant' in queuing theory jargon). The processes might generate a work item deterministically or randomly or their correlation coefficients might be known, so the appropriate M/D/1 or M/M/1 or Kingman's hybrid can be deployed.

Today, moving the data to a shared resource costs appreciable energy and a 'transistors-are-free' attitude would suggest separate FU instances that will not require arbitration and note the multiplexing of the busses also adds capacitance.

Assuming we decide to share a common FU, the various accelerator paradigms lectured in Figures 6.10 through to 6.12 are relevant. These configurations are described in the book. When it comes to sharing them, the stateless nature suggested in this question means there aren't really any further considerations.

Being stateless also makes it fairly easy to virtualise the device under a multi-tasking (time-sharing) operating system like Linux. These aspects are not discussed in the book, but if there were some state, it's worth noting that the custom registers in Figure 6.10b would be context switched at a time slice whereas a bus-connected configuration, as per Figure 6.12, would likely have its own device driver that manages process tags and per-context state swapping when the process tag space runs out.

Q4 Custom Instructions/Accelerators

Consider the following kernel, which tallies the set bit count in each word. Such bit-level operations are inefficient using general-purpose CPU instruction sets. If hardware support is to be added, what might be the best way of making the functionality available to low-level software?

```
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; }
}
```

Answer:

Although the Arm architecture is RISC, some variants include the VCNT instruction that counts bits in words, so using this in the inner loop would be ideal as part of a software solution.

If such an instruction does not exist (or cannot be added as part of an extensible instruction set design chain), a small amount of hardware could easily be added as a coprocessor. A single coprocessor register can be used and operated as follows: it is written with a 32-bit word and the value read back is the ones tally of the word. An atomic exchange instruction is best used to avoid any context swap between the load and store. Alternatively, the read value could be

forced to higher than 32 if the hardware has detected a context swap since the last value was written. Context swaps can be inferred from hardware events, such as an interrupt or write to the VM root pointer register or cache clear instruction, or indeed an ISA might insist that a specific instruction is included in any context swap that makes store-conditional (aka store-exclusive) fail too.

The alternative approach of a separate bus master that collects its own data from the memory might be suitable if a lot of data needs to be processed this way. However, a cache-consistent approach is normally required, and the designer must consider in which cache the data might best be located.

In the first of the two approaches, both the tally and the conditional update of the `maxcount` variable might be implemented in the custom ALU, but most of the gain would come from the tally function itself and the detailed design might be different depending on whether custom instruction or co-processor were used. (Note that Arm has conditional stores which are useful for implementing such a `MAX` function using the everyday ISA.) 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 on a non-superscalar core and hence, if this part of the kernel is placed in the custom data path we might lean more towards the co-processor approach. The separate IP block may or may not use a DMA controller. Given that Arm now has a ones-tally instruction in its main ALU, getting an Arm to move the data into the separate IP block may be a really poor design point!

Q5 Flow Control

Data loss can be avoided during a transfer between adjacent synchronous components by using a bi-directional handshake or performance guarantees. Explain these principles. What would be needed for such components to be imported into an IP-XACT-based system integrator tool if the tool allows easy interconnection but can also ensure that connections are always lossless?

Answer:

This question is about simplex communication (data only moves in one direction) but possibly with bi-directional handshaking.

When data is transferred between a producer (source) and consumer (sink) component, if it is known that one is always ready when the other is, then handshaking is only needed in one direction. In general, both directions are needed. For instance, a fast consumer will always be ready when a slow source generates data: for instance, the consumer might be just adding up the values received. Hence the interface only needs to have a handshake signal in the direction source-to-sink that acts as a data qualifier.

In the extreme, for a synchronous system, if the source and sink are implicitly ready on every clock cycle, no handshake nets are needed and the shared clock is sufficient. [For an asynchronous system, some sort of qualifier is generally needed alongside a parallel data bus to

delimit the bus words.]

IP-XACT allows interfaces to be defined and named protocols to be associated with an interface. For a peer-to-peer (not broadcast) connection, default connection rules connect outputs on one side to corresponding inputs on the other side.

Where a subset of the handshake nets is present on one port, additional rules can easily define default behaviours. For instance, if a sink is always ready, its ready output may not exist and the ready input for the peered source can be tied off to the asserted logic level by the system integration tool.

A more sophisticated approach might compare the advertised rates of generation and consumption of interconnected components, but this cannot guarantee reliability based on average rates, since the peak rates can cause overrun/underrun. Automatic instantiation of FIFOs for balancing this out is not currently within the scope of such tools!

Q6 Pipelined FUs

What is a fully pipelined component? What is the principal problem with an RTL logic synthesiser automatically instantiating pipelined ALUs? A fully pipelined multiplier has a latency of three clock cycles. What is its throughput in terms of multiplications per clock cycle?

Answer:

A fully pipelined FU has an (re-)initiation interval (II) of unity, meaning it can accept new arguments every clock cycle. It will generally have fixed latency of P cycles (pipeline stages) and so the output for a given argument emerges P cycles later. A fully-pipelined multiplier will compute one multiplication per clock cycle regardless of its latency, P.

The synthesizable constructs in the two mainstream RTL languages (VHDL and Verilog) cover combinational and single-cycle sequential logic. A pipelined component can be modelled in these languages with careful coding, using multiple single-cycle assignments. Making sure this is done properly is often achieved by structurally instantiating pre-coded models. Typical such FUs are synchronous multiplier or RAM.

Although these languages support definition and application of functions, but these functions essentially model 'instant' or combinational behaviour: they cannot be directly used for fully-pipelined models because ['*there is no construct available in these languages that describes a function being executed over a duration of multiple cycles*, from *Cycle-accurate RTL Modelling with Multi-Cycled and Pipelined Components*, R Dömer, U Texas, 2016].

On the other hand, HLS tools are able to process complex expressions, such as A[x]A[x*y]/A[y]+. The FUs implementing the subscription (perhaps SSRAM), multiplication and division are likely to have multi-cycle latency and may or may not be fully pipelined.

Q7 Modulo Scheduling

A bus carries data values, one per clock cycle, forming a sequence $X(t) = X_t$ as illustrated in Figure 1.

Also shown is a circuit that supposedly computes a value Y_{t-3}. It uses two adders that have a pipeline latency of 2 and an initiation interval of 1. The circuit was designed to compute the running sum of bus values. Check that it does this or else design an equivalent circuit that works but uses the same adder components.

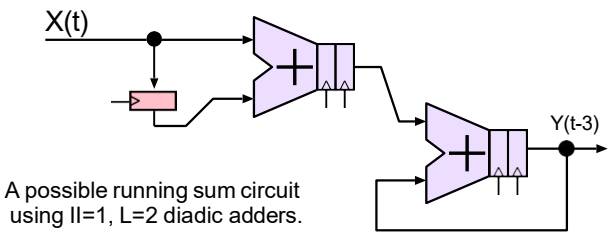


Figure 1: A circuit intended to compute the running sum of streaming data values.

Answer:

Let us define Y_t to be the sum of all previous values of X_t and the current value. Clearly this is Mealy and cannot be delivered from any network or pipelined adders. But generating Y_{t-4} is potentially achievable with the offered circuit since four pipeline stages are clearly visible between X and Y. Let Z_t be the current output from the first adder.

$$\begin{aligned} Z_t &= \mathbf{X}^{-2}(X_t + \mathbf{X}^{-1}(X_t)) = X_{t-2} + X_{t-3} \\ Y_t &= \mathbf{X}^{-2}(Y_t + Z_t) \\ &= Y_{t-2} + (X_{t-4} + X_{t-5}) \end{aligned}$$

This is a recurrence relation. Like differential equations, recurrences can be solved by suggesting a root (i.e. solution) and substituting it into the left and right-hand sides and checking for equivalence. If we now assume $Y_{t-2} = \sum_{i=0}^{t-6} X_i$ we get

$$\begin{aligned} \text{LHS}: \; Y_t &= \sum_{i=0}^{t-4} X_i \\ \text{RHS}: \; Y_{t-2} + (X_{t-4} + X_{t-5}) &= \sum_{i=0}^{t-6} X_i + (X_{t-4} + X_{t-5}) \\ &= \sum_{i=0}^{t-4} X_i \quad \text{QED.} \end{aligned}$$

Hence it does indeed compute the running sum. Whether we call this $Y(t-3)$ or Y_{t-4} is simply a matter of convention and I would suggest the notation used in the above answer is better.

Q8 Operator Identities

Does *strength reduction* help save area, energy or both? Give an expression that can benefit from three different strength reduction rules.

Answer:

The term *strength reduction* is defined in §6.9 as replacing an operator with an equivalent (given by a mathematical identity) that requires less silicon area. An identity equation holds for all values of its free variables, such as $\sin(\theta)/\cos(\theta) = \tan(\theta)$. However, it could also be interpreted in terms of energy and generally also increases performance by shortening critical paths.

The everyday techniques are replacing multiply or divide by powers of two with wiring shifts and replacing multiplication by simple constants by a system of adders. More esoteric examples are replacing comparisons of monotonic functions with comparisons of their arguments.

The following expression illustrates two examples: `(qq*63)/8` can be replaced with `((qq<<6)-qq)>>3)`. A multiplication by -1 example, like the one in the chapter, is `-1*qq` becomes `0-qq`.

Constant folding is also done by the same pass of a compiler: e.g., removal of adding a zero or multiplying by unity. 'Dead code elimination' can also result as a consequence. For instance no trig functions need be computed in the expression `(sin(x)>1.2) ? cos(x): x`.

Q9 Cut-through and Deadlock

Is a NoC that uses store-and-forward elements with cut-through routing a multi-access NoC? Does multi-access lead to more or fewer chances of deadlocking?

Answer:

Section §3.4.1 defines a store-and-forward element to contain a buffer or queue where the entire packet or flit is received before its head emerges from the element. It says a cut-through element supports the head of a packet/flit leaving the element before the tail has fully arrived. §3.3.1 defines multi-access techniques as those relying only on queuing at the source.

Although these terms are clear enough in isolation, many SoC designs are hybrids: they will support cut-through if the output port is idle and use either store-and-forward or source queuing when the output is busy. For lossless operation, flow control back to the source is ultimately needed in all designs.

Hence the first question can be answered either way according to design detail. For instance, if the data bus is sufficiently wide that every flit/packet is sure to be less than the clock, the fabric is using store-and-forward, but it might be using credit-based flow control to avoid overrunning receivers and also using multi-access LAN-style arbitration for the fabric links.

The MAC (access control) protocol for multi-access network fabrics, such as ring and folded bus are indeed always designed to be intrinsically deadlock free. For general mesh networks, deadlock prevention is typically needed, as described in §6.6.3. See also *Autonet: A High-speed,Self-configuring Local Area Network Using Point-to-point Link*s where a spanning tree is superim-

posed at the routing level. In such cases it is arguable whether the deadlock avoidance system is part of the network or is a set of rules applied to users of the network. Having a deadlock-free NoC of whatever type does not avoid deadlock overall: state machines associated with higher layer protocols can always deadlock if poorly designed.

It is important to note that deadlock avoidance is non-compositional: composing automata that are themselves deadlock-free can lead to deadlock. For instance, at a very high level, I might need to get my phone out of the locked boot (trunk) of my car to get run the Android app that unlocks the boot! The running Android app will use the NoC inside the phone SoC and hence that NoC is part of the deadlocking cycle.

Q10 Loop-carried dependencies.

Does either of the following two loops have dependencies or anti-dependencies between iterations? How can they be parallelised [48]?

```
loop1:  for (i=0; i<N; i++)  A[i] := (A[i] + A[N-1-i])/2
loop2:  for (i=0; i<N; i++)  A[2*i] = A[i] + 0.5f;
```

Answer:

These are taken from [48] J. Liu, J. Wickerson, and G. A. Constantinides *Loop splitting for efficient pipelining in high-level synthesis* in 24th Annual International Symposium on Field- Programmable Custom Computing Machines, pages 72–79. IEEE,2016.

loop1: Assume N is even. If the loop only went up to N/2 (a more typical example), there are no loop carried dependencies since the second half of the array is only ever read and each location in the first half is read before it is written. There are no anti-dependencies between loop iterations either (we don't need to defer any write until after the location has been read since the causality of the assignment is sufficient: i.e. the r.h.s. must be evaluated before the l.h.s. can be assigned.) Hence it would be so-called *embarrassingly parallel*.

However, loop1 continues into the second half of the array where each operation reads one of the outputs from the first half of the array. Hence there is a (long distance) loop-carried dependency the way the code is written. Clearly this can be nicely recoded to iterate up to N/2 and to do both the assigns to `A[i]` and `A[N-1-i]` in the loop body. The recoded form is embarrassingly parallel with degree N/2.

If N is odd, special code for the middle value needs to be added, but this can also be run in parallel.

loop2: Figure 2 of the paper shows the dependencies are fairly complex. The pattern is called 'non-uniform' because it varies from one iteration to the next. However, there is structure and the presented technique in the paper, or just pen-and-paper, can find a parallel speedup of about 6 for N=70.