Modern SoC Design on Arm
End-of-Chapter Exercises

Q1 Roundtrip Delay

What is the principal reason that protocols that fully complete one transaction before commencing another have gone out of fashion? Estimate the throughput of a primitive MSOC1-like bus protocol implemented with modern technology.

Answer:

The principal reason they are no longer suitable is that, with increasing clock frequency and conductor resistance, net delay across a chip has become more than a clock cycle. This means that system busses need to regenerate long distance digital logic signals, such as with a D-type flip-flop every couple of millimetres. These D-types create a delay line, limiting the minimum time between a request and a response. Sometimes a response is needed before the next request can be made. But mostly this is not the case. Therefore, it is now completely normal to use bus protocols where transactions overlap in the time domain with multiple transactions in-flight.

The MSOC1 protocol in the book used combinational logic. If the operating area was within one millimetre or so and the clock frequency was sub-gigahertz, the throughput could be unity (one operation per clock cycle). For a data bus width of 32 bits and a 500 MHz clock, this is 2 GByte/s. But over larger distances and at higher clock frequencies, the protocol is completely impractical. To cross the whole chip, there and back, with combinational logic, regenerated with inverters where necessary, the clock frequency achievable would be well under 100 MHz. So less than 3.2 Gbit/second (400 MByte/second).

Q2 Split Bus Energy Use

What affects interconnect energy consumption as the number of channels that make up a port is increased from two (for BVCI) to five (for AXI)?

Answer:

The average toggle rate of the nets within the port is the principal contributor to energy use by such a bus. In AXI, having separate read and write channels divides the number of transactions per channel (and hence transitions per net) down with respect to BVCI. On the other hand, there are more channels. The total number of nets is not significantly changed and the useful information flux through the various schemes is necessarily the same. The number of net transitions (half toggles) on the two address busses of AXI could be slightly lower than on the combined address bus of BVCI if there is correlation between successive read (and ditto write) addresses that would be broken up with a shared bus: e.g., in a block copy. This could marginally reduce energy use.

Q3 Cache Coherent I/O Devices

Why is a mix of coherent and non-coherent interconnects always found on a SoC? Why are some peripheral devices connected to a special purpose bus?

Answer:

A cache-coherent interconnect carries sufficient information for caches to be kept up to date. For instance, using the MESI protocol, the interconnect must carry evict messages to make sure a dirty datum is stored in precisely one place. The additional complexity is generally not relevant or warranted for peripheral busses. Many I/O devices have low throughput (e.g., UART) or do not generate bus activity themselves (they are always responders and not initiators). These devices are generally connected to simpler and lower-frequency busses using a bus bridge to/from the coherent central zone.

For higher-performance I/O devices, typically using DMA, there is often no cache between the device and the DRAM (except within the DMA controller itself). A hardware overhead arises when connecting the DMA mechanisms to the consistency mechanisms (e.g., so that locations updated by DMA are invalidated in processor caches). So instead, a software solution can be more efficient: the device driver issues cache invalidate commands for regions known to have been updated by DMA.

Q4 Regenerating Bus Protocols

Sketch circuit diagrams for a registered pipeline stage inserted into an AXI channel and a CHI channel. What design decisions arise in each case and what effect do they have on performance and energy use?

Answer:

The CHI channel is very easy to augment with a pipeline stage for signal restoration in either or both directions. There are no consequences for the protocol semantics which is one, main advantage of credit-based flow control. On the other hand, performance in terms of saturated throughput can start to decrease if there is insufficient credit and receiver buffer space to accommodate the additional round-trip delay. The circuit diagram is simply a broadside register across all the forward nets and/or in the credit return channel. This is shown in Figure 3.25.

Data on an AXI channel is qualified by the conjunction of the valid and ready signals. If all of the nets in one direction are put through a broadside register, the handshake signal in the other direction will be displaced by one clock cycle on one of the two sides and the protocol is destroyed: the two ends will have different ideas over which words are valid. If both directions are registered, the effect is twice as bad. Hence a simple broadside register will not work. Instead, a synchronous FIFO structure is needed, as in Figure 6.8.

Buffering or retiming is needed to get a signal a long way. (The logical effort approach to regeneration offers a theoretically optimal way of doing this, but it can just be designed using rules-of-thumb validated by simulation.) Re-timing using edge-triggered flip-flops uses more energy than just buffering (using an even number of spaced-out inverters), but better compensates for net skew.

Given that regeneration is needed, it could be said there is no energy increase with either scheme since the system would simply not work (meet timing closure) without regeneration.

With credit-based flow control, if increasing the receiver buffer size would be needed, twice as many D-types would be instantiated as is needed for standard-synchronous handshakes like AXI, since the data is being registered both in the channel and in the receiver. But in some cases there may be sufficient receiver capacity already.

FIFOs without Mealy paths can insert bubbles. To assist with timing closure, Mealy paths must be avoided, meaning that either bubbles are tolerated (increasing latency and reducing throughput sometimes) or the FIFO stages must have additional capacity (e.g., store two datums instead of just one).

Q5 Multi-beat Transactions

Sketch the circuit for a bus width converter for an AXI channel if the same clock frequency is used on each side. What are the differences from credit-based flow control? When credit-based flow control traverses a bus width changer, what is the most sensible meaning for a credit token?

Answer:

As mentioned in the last answer, an AXI channel uses the standard synchronous handshake. The width converter may halve or double the main bus width and pass on the further signals in the same quantity as they arrived. How many further signals there are depends on which channel and which variant of AXI. Nonetheless, the basic design is the same in all getting wider designs and ditto the getting narrower situations. Also, there may be a 'last' signal on the narrower side where this is not needed on the wide side. The converters are variations of a basic FIFO design, so the amount of internal storage and presence of Mealy paths is a design parameter.

Here is the skeleton Verilog for a simple D16 to D32 channel expander. It has three states: empty, half full (16 bits has arrived) and full.

```
module d16tod32_channel_expander(input clk,  // Sketch of the basic structure
   input [15:0] din,             output [31:0] dout,
   input din_valid,              output dout_valid,
   output din_rdy,               input dout_rdy);

   reg [1:0] state; // 0=empty, 1=half, 2=full.
   reg [15:0] lo_reg, hi_reg;

  always @(posedge clk) begin
    if (state==0 && din_rdy && din_valid) begin
      state <= 1;
      lo_reg <= din;
      end
    if (state==1 && din_rdy && din_valid) begin
```

```
      state <= 2;
      hi_reg <= din;
      end
    if (state==2 && dout_rdy && dout_valid) begin
      state <= 0;
      end
  end

  assign dout = { hi_reg, lo_reg };
  assign dout_valid = (state == 2);
  assign din_rdy = (state != 2);
```

With credit-based flow control, both ends must share the same idea of how much data (how many bits) are guarded by a token: in other words, the flit size. When a flit crosses a width converter, although the number of clock cycles needed to convey it may change, the amount of data is unchanged: the converter is an identity function, semantically speaking. Hence, in a baseline approach, the meaning of a token is unchanged.

Q6 Virtual Circuit Buffering

A NoC uses static TDM to separate VCs on a link with the schedule fixed at tape out. Should the receiving link have a shared buffer pool or a pool that is statically partitioned for use by different VCs?

Answer:

As explained in section 4.3.3, a shared memory pool is more efficient than separate pools owing to statistical multiplexing gain. Hence there is always a preference to use a shared pool. But this does lead to more complex control logic and multiplexing paths, especially when the pools hold a few words only — a case where the overheads of a RAM do not outweigh the use of a register file.

The use of separate VCs is commonly used as a simple means of avoiding deadlock. Unrestricted sharing of the pool will defeat this: messages may be blocked owing to lack of buffer space where the buffer space that is being used by another VC cannot be freed up owing to an outer cyclic dependency. The solution is for the pool controller to statically limit the maximum amount of store allocated to a specific VC to a pre-assigned quota. These quotas must be computed by global analysis of the system.

This is a bit of a trick question: whether the links are statically or dynamically scheduled is orthogonal to the buffer allocation policy.

Q7 Dynamic Bandwidth Allocation

Another NoC uses dynamic TDM. Additional nets convey a VC number that identifies the data on the remainder of the data nets. Discuss the likely performance and energy differences compared with static TDM. (You should be able to improve your answer after reading the next chapter!)

Answer:

A static allocation or scheduling does not adapt at run-time and is likely to perform badly in situations different from what was modelled at tape-out time. The advantage of static scheduling is typically simplicity, leading to silicon area and energy savings. For virtual circuits on a NOC, the static schedule is generally based on a global modulo counter, or many synchronised, local copies thereof. Specific count values are pre-allocated to particular VCs. If there are two VCs, request and response, in the simplest system, the counter needs one bit and alternate cycles denote request and response slots. Words of flits on different VCs can also be interleaved: this does not change throughput or average latency but reduces jitter (random variations in latency).

Dynamic TDM, where flits are tagged with VC number, uses additional energy conveying the tags. The circuitry at each end is also more complex, with an arbiter being required to select which VC to service next at the sender, and more complex de-multiplexing logic at the receiver.

Overall, dynamic TDM should give better performance owing to statistical multiplexing gain, but also use more energy, mainly owing to the additional information conveyed.

Q8 Round Trip Time Again

For what types of application does NoC latency affect system throughput?

Answer:

[This question is exactly analogous to wide-area networking questions, where longer round-trip delay can slow down distributed applications despite there being plenty of bandwidth available.]

Where work can be generated without waiting for a previous response, latency is generally unimportant. Where the next work item depends on the response from a previous one, latency matters. Reading memory over a high-latency link serves as a good example. Where the access pattern is predetermined (e.g., for matrix multiplication) the latency should not be an issue. On the other hand, following a linked list or tree walking suffers badly from NoC latency, since the next request can be hard to predict and is really needs to deferred until the previous result is to hand.

Q9 Programmers Doing Consistency

What are the advantages of having fully automatic hardware support for memory coherency compared with leaving it up to the programmer to insert special instructions?

Answer:

Fully automated cache coherence is the norm for today's mainstream computers. The application programmer can assume that any data they read is the most recently written data to that location by any member of the coherence group. It is argued that this is a highly desirable hardware feature. However, it is now widely appreciated that fence instructions must be manually

inserted by programmers on machines with relaxed memory ordering to preserve sequential consistency. Hence, having reliable memory coherence only at fence points is a possible design point with a low amount of new baggage. Moreover, optimising compilers for C and related languages will hoist loads and stores, leading to an inconsistent view for the programmer even though the hardware was 'perfect'. Again, fences implied by synchronisation primitives are today's answer.

Additionally, the class of user-level algorithms that benefit from cache coherence is small. Much inter-core communication is performed by user-space messaging libraries, perhaps provided by the O/S or being built into the VM or run time system of a high-level language. Alternatively, a lot of inter-core communication uses locked, shared data structures. Systems that only guarantee cache consistency at an unlock point are the basis for hardware-based transactional memory paradigms.

Overall, this is a slightly controversial question. The same points can be made, to a weaker extent, in the context of caches versus scratchpads. It is undeniably important to have a mixture of memory technologies to achieve a sweet spot in the energy/performance envelope, with dynamic migration of data between memory technologies (i.e. different caches), the overhead of additionally keeping them consistent is not large and not having it is too disruptive to computing tradition.

NB: This question is about user-space applications. The discussion for I/O traffic and device drivers was discussed earlier.

Q10 An array of mutexes

A C programmer writes `pthread_mutex_t locks[32]`. A friend says this will have very poor cache performance. Why might the friend say this? Are they correct?

Answer:

Locks are used for shared data structures and so they are commonly accessed by different cores. The size of a lock can vary quite a lot, depending on their implementation on a particular system. A simple mutex might consist of a 32-bit word allocated by the programmer. This is possible when the further information, such as the list of threads waiting on it, is held in the thread scheduler instead of in the lock itself.

A cache line might accommodate four 32-bit locks. For this to happen, they need to be allocated side-by-side in memory, as with the programmer's array of 32 locks. The problem that arises is one thread (or core) may be rapidly acquiring and releasing one of the locks with the majority of operations not seeing any contention from the other threads. The locking overhead will be minimal. But if another thread is doing the same with another of the locks that share the cache line, the line will be flipping between cores at a high frequency. A latency penalty will arise, especially if the cores are on different chips.

The solution is for the programmer to use an array of structures where each structure is the size of a cache line. The lock uses part of the structure and the rest is padding. For instance,

```
    struct { pthread_mutext_t lock; char [48] padding_; } locks[32];
```