Modern SoC Design on Arm
End-of-Chapter Exercises

Note: The exercises in this chapter are somewhat different from those in other chapters, since they assume a broad basic knowledge of processor architecture and assembly language programming. They may require materials not presented in the book.

Q1 Hazards

Give examples of assembly language programs for a simple in-order processor that could suffer from each of the following problems and describe hardware or software mitigations: (i) a control hazard, (ii) a hazard arising from the ALU being pipelined and (iii) a load hazard, even though the data are in the cache.

Answer:

The following fragment can suffer a control hazard in a pipelined implementation. Assuming no branch predictor, or predicted not to be taken, the add to register 4 will have been inserted in the pipeline before the predicting condition for the branch has been determined. This can lead to the add being executed when it should not have been. This is known as the 'branch delay slot instruction'. Solutions include specifying that the instruction **is** executed as part of the ISA definition (software then makes use of it or puts in a NOP) or making sure the effect on R4 is suppressed in the write-back stage of the pipeline.

```
     CMP R2, R3 // Compare registers
     BLT foo    // If R2<R3 goto foo
     ADD R4,=1  // Immediate add of one to R4
foo: SUB R5,=1
```

The following fragment can suffer an ALU pipelining hazard. Multiply (and certainly divide) can take more clock cycles than ADD. Hence the multiplication result is not ready when it is needed in the next instruction and the old value of R2 could be used. (If we assume a forwarding path normally solves this problem for simple ALU operations, the same path will not help for an extended latency ALU operation). The software solution is to define in the ISA spec that R6 would not have its new value until later. Such a specification can overly constrain the design of future processor variants, so is avoided in today's mainstream architectures. A pipeline stall will be used in a simple hardware implementation. A complex, out-of-order implementation would instead often be able to schedule other instructions while the multiplication completes.

```
     MUL R2,R4,R5    // Update R2 with the product of R4 and R5
     ADD R2,R2,R6    // Add the contents of R6 to R2
```

The following fragment can suffer a load-delay hazard in a pipelined implementation. If the front-side of the data cache has a 2-cycle latency, the value will not be in R2 when it is needed as an operand for the addition. Hardware solutions include adding a stall (or possibly a forwarding path, but that is more typically used for ALU results) and the software solution is to define in

the ISA spec that R6 would not have its new value until the instruction after the 'load delay slot'.

```
LOAD R6,[R1]    // Fetch a value from memory into R6
ADD R2,R2,R6    // Add the contents of R6 to R2
```

Q2 Misbehaving Cache

If the front side of a cache has the same throughput as the back side, owing to the back side having half the word width and twice the clock frequency, for what sort of data access pattern will the cache provide low performance?

Answer:

Under normal circumstances, temporal and spatial locality effects mean that far less back-side bandwidth is needed than front-side. An equality of bandwidths between the front side and the back side is perhaps unusual and certainly should not normally be relevant: a good cache achieves a hit rate above 90 percent and so the ratio of bandwidths is not an important aspect. It is, however, good if the back side can be served with low latency when a miss occurs and lower latency arises from higher bandwidth with fixed-size messages, such as cache lines.

If the front-side experiences low spatial locality, the back side will be fetching up to twice the necessary amount of data and its bandwidth will be effectively halved.

Given that the bandwidth ratio is largely irrelevant (a red herring), the question is now asking when does a cache not perform well. As described in the chapter, caches miss for sharing, compulsory or capacity reasons. Capacity problems are exacerbated by aliasing issues. An aliasing problem can arise with any directly mapped aspect of the allocation policy where the access pattern happens to step with the directly mapped strath and does not repeat within the associative set size. Sharing evictions arise when DMA or some other core is writing to the cached data: this can sometimes be avoided by moving work such that it goes through the same cache (e.g., hyperthreading on one core).

Q3 Hyperthreading

If a super-scalar processor shares FPUs (floating-point units) between several hyper-threads, when would this enhance system energy use and throughput and when will it hinder them?

Answer:

A super-scalar processor is defined to be able to execute more than one instruction per clock (IPC>1.0). Hyper-threaded cores have more than one user context (register file) sharing a pool of FUs, such as floating-point units (FPUs). If two cores sharing a set of FPUs are running highly diverse workloads, they are unlikely to both be using floating-point at the same time. For instance, one might be running a compiler that uses hardly any real numbers and the other doing neural inference that has a heavy floating-point load. This will lead to good overall performance and energy is saved owing to there not being a set of FPUs that is sitting idle. Idle FPUs could consume static energy, or could be turned off, but even then, their area will be extending net

lengths in general.

Performance will be degraded if both cores are running code this is heavy in floating-point instructions.

Q4 Serial vs. Parallel I/O

Why has serial communication been increasingly used compared with parallel communication? Compare the parallel ATAPI bus with the serial SATA connection in your answer.

Answer:

Electronics has been getting ever cheaper and the clock rates achievable in CMOS have been getting ever higher. Also, copper for making wires is getting more expensive. Today, a radio link can be cheaper than a multi-core copper cable: a fact that engineers of the 1960's would find totally amazing! A parallel interface used to be simple-to-design and easy-to-make and was historically the preferred technique. A good example is the Centronics-style printer port.

At higher clock rates, the problem of skew in parallel cables becomes excessive. As the link length approaches the wavelength of the clock (e.g., 1 metre at 150 MHz given $\epsilon_r = 4$ for PVC), different delays for each signal raises problems. Additional electronics at the end of each conductor becomes preferable to manufacturing tightly matched electrical conductors.

The first-generation ATAPI bus for disk controller connection to motherboards used low-quality parallel wiring. To increase its performance, expensive ribbon cables were developed and the number of ground connections needed to be increased. This could not easily be scaled further to higher data rates for the reasons just stated. Meanwhile, the energy and area costs of SERDES technology were falling per bit per second: hence SATA.

Q5 Virtual and Physical Cache Tags

Assume a processor has one level of caching and one TLB. Explain, in as much detail as possible, the arrangement of data in the cache and TLB for both a virtually mapped and a physically mapped cache. If a physical page is mapped at more than one virtual address, what precautions could ensure consistency in the presence of aliases? Assume the data cache is set-associative and the TLB is fully associative.

Answer:

A reasonable level of detail would include a list of fields in a feasible TLB entry and a sketch of Figure 2.8 from the book. The TLB entry has two halves: the associative tag and the data stored.

Assuming a 32-bit address space with 4Kbyte pages, the tag will have at 32-12=20-bit virtual address tag. TLBs are normally fully associative, so no tag bits disappear owing to a directly-indexed dimension. Some architectures extend the virtual address with 'process tag' dummy bits, enabling entries to persist over context swaps. The data will contain the physical frame number and various flags. The bottom 12 bits of the virtual address are the offset within a page. These do not need to be stored since they are unchanged during mapping.

There will be flags for validity, where a page has been written or accessed, access permissions and dirty flags for written and accessed. A dirty entry needs to be written out to the page tables on eviction. It can be dirty since it was not marked as accessed or modified when loaded.

The layout of the TLB entry does not depend on which caches are virtually tagged.

Many processors follow the design of Figure 2.8, using a virtually mapped first level data and instruction caches (at least in the directly mapped axis) and physically mapped caches beyond the MMU. This balances the logic delays into the L2 comparators.

It is undesirable to increase the cache line size or page size, so to support larger L2 caches in modern designs we can either increase the number of ways (which is costly because of replicated tag logic and of little benefit owing to the birthday distribution) or the directly-mapped dimension (which is cheap).

But if the page size is kept the same while increasing the directly mapped dimension by, say, one bit, then the least-significant bit of the TLB output (physical frame number) is now used as a new m.s.b. row index in the L2 cache, doubling the cache size.

If the same physical segment or page is mapped more than once for one process/core then it must be at different virtual addresses, and these might differ in the least significant bit of the physical frame number and hence address separate directly mapped indexes since that bit has now been used for that dimension. The problem is now that the required tag may exist in the cache at the row with the inverted address bit. Hence a proper hit will be missed. Both reads and writes that should hit may miss, which is not a critical problem for reads, but where one copy is dirty, critical inconsistencies arise.

A software workaround is to make sure the operating system eliminates this problem in its physical page allocation policy. This is trivial and harmless. The procedure is known as colouring. With one bit, we divide physical and virtual pages into two equal-sized sets, black and white, depending on the value of the l.s.b. of the frame number. Naively, we then ensure that we match colours in all virtual to physical translations, making the bottom bit in all TLB translations the identity function.

Note: this colouring does not increase the effective page size (which is what we were trying to avoid) and so does not increase internal fragmentation of pages or decrease the resolution of the TLB. It does possibly lead to the case where the O/S has run out of one colour of pages before another, but this is a negligible penalty if you work out the statistics, given millions of pages and two or four colours.

The O/S only needs to apply the colouring policy to pages that are likely to be multiply mapped (which will typically be a segment policy), but hypervisors may also use such tricks to better isolate cache competition between different virtual machines.

Q6 Interrupt Routing

What are the advantages and disadvantages of dynamically mapping a device interrupt to a processor core? What should be used as the inputs to the mapping function?

Answer:

The pattern of wiring of interrupts from devices to cores can be fixed in silicon or made programmable. A programmable matrix can be set once at boot time, which we would call static, or can be adjusted at run-time, which is dynamic. Dynamic routing could direct an interrupt to a core which is currently idle, which is an advantage for load balancing. But for periodically occurring interrupts (frequent ones), it is helpful to repeatedly direct that interrupt to the same core to avoid cache thrashing.

A SoC operating system commonly maintains loading statistics for its cores, so an O/S daemon could possibly reprogram the interrupt routing matrix on a regular basis (e.g., 10 to 50 Hz). This would probably be overkill. A common approach is for the routing matrix to maintain round- robin state to allow it to spread interrupts over a set of enabled cores in an even manner and to avoid routing an interrupt to a core that is already servicing an interrupt.

Q7 Flash Everywhere?

If a new variant of a microcontroller uses a single non-volatile memory technology to replace both the static RAM and mask-programmed ROM, what are the possible advantages and disadvantages? Is this even possible?

Answer:

Conceivable advantages are that having one type of memory makes the design simpler and gives a statistical multiplexing gain whereby it is impossible to run out of one type of resource while another has capacity. The idea is feasible for certain niche applications.

ROM, RAM and Flash cells have different densities, so a device that currently stores a lot of information in ROM could become more expensive if that data is instead stored in never-overwritten Flash. The different memory technologies support different numbers of write cycles (ROM none, Flash bounded, RAM unbounded). Applications that keep a rapidly changing data structure in memory or transiently buffer a lot of data need to use RAM. Applications that require an incorruptible 'reset to factory settings' must either use ROM for those settings or else have at least one write-once 'fuse' to lock down parts of the non-volatile, writable memory.

Q8 Primary vs. Secondary Store

Some PC motherboards now have slots for high-performance non-volatile memory cards. How can these be used for primary or secondary storage? Should computers continue to distinguish between these forms in their architecture?

Answer:

The question is referring to Flash memory and the distinction between DRAM, NVMe (Non-Volatile Memory Host Express) sticks and SSDs. The NVMe sticks have a PCIe interfaces instead of the SATA used on solid state disk drives (SSDs) and can have a data transfer rate 10 times faster than the 500 MB/sec peak of SATA.

The least disruptive option is to make the memory stick appear as another disk drive and recommend to users that they store commonly accessed files there for high performance. The distinction between primary and secondary storage is whether the main processor can directly read or write it. There is a strong expectation that file systems on secondary storage are very largely preserved over system crash and reset: if this non-volatile memory is connected as primary storage and a program crashes in supervisor mode, there is a chance that parts of the file system is undesirably overwritten.

Another possible use is as traditional swap space to augment the effective amount of primary storage. But using it for swap space may run into write wear level issues.

When memristive cross-point memory first became available, the idea of using it instead of DRAM was widely mooted, leading to non-volatile primary storage and instant power-up booting. Although still not widely used, its legacy has been new form factors (especially M.2) for low latency connection to Flash memory, which intrinsically has very high read speeds that are throttled by SATA when accessed in SSD format.

Q9 Microcontroller Programming

Briefly describe the code and wiring needed for a seven-segment display to count the number of presses on an external push button accurately. Note that mechanical buttons suffer from *contact bounce*. Use polling for your first implementation. How would you adapt this to use a counter/timer block and interrupts? What are the advantages of this button and display application?

Answer:

With today's technology, the cheapest implementation using off-the-shelf components is likely to use a microcontroller (perhaps an Arm Arduino). The push button and the LEDs can be directly connected to PIO pins on the microcontroller (although PNP high-side transistors might be needed for the LED anode commons). The display will be scan-multiplexed with one digit illuminated in a time slot of, say, 5 ms.

In the polling implementation, the processor might use an eternal outer loop that contains, at one point, a tight delay loop such that the outer loop frequency is 200 Hz. This same loop can read the push button and see if its value is the same as that last read. Any set of two or three consecutive reads of the same button value can be considered the new denounced value. The counter variable needs incrementing on every other change of value (i.e. the down press).

The processor can be freed for other work and can avoid wasting time in a delay loop by programming a timer to give a 200 Hz interrupt. All of the work of the outer loop can be put in the ISR. Technically, the switch will still be polled. A further mootable enhancement would be to connect the switch input to an interrupt. This is not necessarily going to be a useful step forward since the rate of interrupts might go up overall if the switch is very bouncy (e.g., bounces of 500μs) and the work of the interrupt routine will be to effectively to aggregate this high data rate down to the sort of sample rate (around 200 Hz) which is ideal for debouncing switches without making them seem sluggish.

In terms of energy use, using timer interrupts will be more efficient that sitting in a delay loop since the processor can sleep while waiting for interrupt. However, the energy saved is likely to be negligible, saving micro-watts. This must be compared with the LED display that would be using a fair fraction of a watt. [Using the back EMF from an inductor to drive the LED segments would be the best step towards lower energy — such circuits are commonly used in solar garden lights, whereas microcontroller-driven LEDs generally waste at least half their energy in current-limiting resistors.]

Q10 Video Capture and Display

A SoC is required to have frame stores for video input and output. Could these follow essentially the same design with minor differences? Would it be sensible to support a number of dual-purpose frame stores that can operate as either an input or output?

Answer:

Yes and yes. A frame store contains a memory location for each pixel and a counter that scans through them to create a raster with horizontal and vertical flyback. The circuit structures for video capture and video render are very similar: it comes down to whether the memory is being written or read. A minor issue of complexity is that a video output system can easily be clocked from the system clock whereas digital video being received would likely come with its own clock and ultimately require a clock domain crossing bridge at some point.