Module 4F14: Computer Systems

Solutions to 2021 Tripos Paper

1. **Instruction set architectures, datapaths, pipelining**

(a) A load-store instruction set architecture is an example of a general purpose register (GPR) architecture where the operands of arithmetic and logic instructions must be located in registers, not memory. If a memory access always took one clock cycle, then there would be little downside to allowing arithmetic and logic instructions to operate directly on operands in memory. However, over the years CPU speeds have increased significantly faster than memory speeds, and caching strategies have evolved to address this discrepancy. A consequence is that memory accesses are now highly unpredictable: there may be no latency for a top-level cache hit, or a small latency for a lower level cache hit, or a longer latency for a main memory access, or an enormous latency for a page fault. In this context, a load-store architecture is attractive in that this uncertainty is factored out from arithmetic and logic instructions, with typically comprise around 50% of the instruction mix. Optimal scheduling of these instructions, whether static or dynamic, therefore becomes much more tractable. In contrast, a non-load-store architecture has to allow for arithmetic and logic instructions stalling for an unpredictable number of cycles while waiting for memory, which makes them much less easy to schedule optimally. From a historical perspective, then, it is not so much the case that load-store architectures were specifically invented, but rather that load-store operations were gradually removed from arithmetic and logic instructions in response to the evolving characteristics of memory systems. [25%]

(b) The `add` instruction is what discriminates between these three ISA classes. It is clear from the code segments that the operands `A`, `B` and `C` are all in memory. For a stack ISA (the first code segment), operands are implicit and taken from the stack, so the `add` instruction has no explicit operands. Instead, the operands `A` and `B` are first pushed onto the stack and the result `C` is then popped off the stack. For the accumulator ISA (the second code segment), one operand is implicitly the accumulator, the result also ends up in the accumulator, and the `add` instruction takes just one further operand (`B`). The other operand `A` is loaded into the accumulator before the `add`, and the result `C` is transferred from the accumulator following the `add`. The third code segment is the load-store architecture, where all operands are explicit and the `add` instruction involves no memory access. [20%]

(c) The datapath can already add a register to a 16-bit sign-extended constant within the instruction: this is how load-store instructions calculate the memory address. So `addi` will work with the datapath as it stands. Furthermore, no extra control signals are required, just a new combination of the current signals:

Signal	Setting
MemRead	Low (no memory access)
MemWrite	Low (no memory access)
ALUSrc	Sign-extended instruction[15-0]
RegDst	Instruction[20-16]
RegWrite	High (write register)
Branch	Low (replace PC with PC+4)
MemtoReg	Low (write ALU output to register file)
ALUOp	00 (add independent of instruction[5-0])

[15%]

(d) With the modified instruction set, load-store instructions no longer use the ALU to calculate the memory address: instead, the address comes straight from the register file. The data memory address input would need to be connected to the "read data 1" output of the register file instead of the ALU output.

The longest path through the datapath is now for the R-format and `lw` instructions, which both require $2 + 1 + 2 + 1 = 6$ns. The datapath could now process $10^9/6 = 166.7 \times 10^6$ instructions per second, at the expense of more instructions.

Consider a program which executes n instructions, including kn offset load-store instructions. The time to execute this program on the original datapath is $8n$ nanoseconds. With the new datapath, after the assembler has translated the pseudo-instructions we have $(1 - k)n + 2kn = (1 + k)n$ instructions, taking $6(1 + k)n$ nanoseconds to execute. For a positive impact on performance, we require $6(1 + k)n < 8n$ or $k < 1/3$. So we require fewer than 33% offset load-store instructions for the modification to be worthwhile.

The traditional way to pipeline the stripped-down MIPS datapath is with five stages: instruction fetch (IF), instruction decode and register fetch (ID), execution and effective address calculation (EX), memory access (MEM) and write back data to registers (WB). Since the speed of the pipeline is limited by the slowest stages, one instruction can be issued every 2 ns. Hazards unresolved by data forwarding or compiler optimisations may slightly reduce the processing rate.

With the instruction set modification, no instruction will ever need to go through both the EX and MEM stages, so these parts of the datapath can be combined into a single EX/MEM pipe stage: load-store instructions will access memory, other instructions will use the ALU. Our new, four-stage pipeline has reduced latency but offers no throughput advantage: it can still accept only one instruction every 2 ns. The one advantage might be the reduction in data hazards: with data forwarding, the result of a `lw` can be available at the ALU input for an instruction following immediately behind. However, it is unlikely this will compensate for the extra `addi`'s introduced by the pseudo-instruction translation.

[40%]

Assessors' remarks: This question tested the candidates' knowledge of instruction set architectures, datapaths and pipelining. It was generally well answered, the only weak link being the second part of (a) where candidates struggled to extrapolate their knowledge to

speculate how load-store operations might have been gradually removed from arithmetic and logic instructions in response to the evolving characteristics of memory systems. Answers to all other parts of the question were generally good, though a few candidates failed to realise that the modification in (d) allows *all* instructions to run faster on the unpipelined datapath (i.e. a faster clock speed), not just load-store instructions.

2. **Parallel processing, caches, DMA**

(a)

MIMD, Multiple Instruction Streams Multiple Data Streams. A computer classification in Flynn's taxonomy of parallel processing machines. Multiple uniprocessors connected on a single bus or via a network. The most general form of parallelism.

SMP, Symmetric Multiprocessor. A type of single address space multiprocessor in which accesses to main memory take the same amount of time no matter which processor requests the word and no matter which word is requested.

UMA, Uniform Memory Access. Means the same thing as SMP.

NUMA, Nonuniform Memory Access. A type of single address space multiprocessor in which some memory accesses are faster than others depending on which processor asks for which word. [10%]

(b) A small number of processors can be connected together as in Machine A. Since each processor has its own cache, the single bus and memory system can serve the needs of all the processors, as long as there are not too many of them (up to a few tens of processors). This is a SMP.

With more processors, the single bus and memory system becomes a bottleneck, and the need for a physically longer bus also reduces the bus's bandwidth and increases its latency. So architectures like Machine B tend to be used for larger MIMD machines. The memory is distributed amongst the nodes, so local processor-memory traffic can proceed at a high rate, independent of the number of processors. Inter-processor communication, however, is over a network and slower than in a single bus design. [10%]

(c) Machine B can use either shared memory or message passing. Just because the memory is distributed doesn't mean it cannot be shared: the physically separate memories can be addressed as one unified address space, albeit with nonuniform memory access. Alternatively, each processor's memory can be completely private and inaccessible to remote processors: the processors will then have to communicate by message passing. [10%]

(d) Increasing the cache block size can lead to a higher miss rate in SMPs if a write-invalidate coherency protocol is used. Suppose processor A writes word X. Also suppose that processor B's cache contains a copy of the block containing X: this block is invalidated when A writes. Now suppose processor B wishes to read word Y, which is distinct from X but in the same cache block. The read will miss, since the cache block has just been invalidated by A's write, even though word Y was not written by A. This cache miss would not occur with one-word blocks. This phenomenon is known as *false sharing*. [20%]

(e) We will assume that the multithreaded software is correctly written with appropriate synchronisation, so that Thread B does not attempt to read the shared memory until the I/O has completed. But there is still plenty for the hardware and operating system to do, to ensure that Thread B sees the updated data.

First, note that the question asked about measures to ensure that the CPU is not unduly loaded by the I/O. This implies direct memory access (DMA). With DMA, the CPU hands control to a DMA controller, which deals with the individual bus transactions between the device and memory. Once the transfer is complete, the CPU is interrupted. The CPU then checks whether the transfer was completed successfully or whether there was an error.

DMA poses challenges for both virtual memory systems and caches. Starting with virtual memory, the DMA controller is supplied with a starting address and a number of bytes to transfer. But should the starting address be virtual or physical? If virtual, then the DMA controller will need extra hardware to store the necessary page table entries and perform the translations. If physical, then care must be taken to ensure that DMA transfers do not cross page boundaries, since contiguous virtual pages do not generally map to contiguous physical pages. Whichever approach is taken, the operating system must cooperate by not moving pages around while a DMA transfer involving that page is in progress.

Moving on to caches, there is a potential problem when transferring data from disk into memory, as in this example. There may be a copy of this chunk of memory in the caches of both Cores 1 and 2. So when Thread B, on Core 2, next reads from the shared memory, it may get the old value (from the cache) and not the new value (as updated by the DMA controller). There are three ways round this. One possibility is to route all DMA activity through both caches, though this is expensive and very wasteful of cache space, since the processor rarely needs to see all the I/O data immediately and, in the meantime, useful data has been displaced from the caches. The second option is to have the operating system invalidate the caches for an I/O write or force write-backs for an I/O read. This sort of *cache flushing* requires minimal hardware support but is inefficient, since the whole cache is affected even if only one block overlaps with the DMA activity. The final, most complex option is to provide mechanisms to selectively flush individual cache entries. [50%]

Assessors' remarks: This question tested the candidates' understanding of parallel processing hardware and I/O protocols. Answers to (a) and (b) were very good, with all candidates demonstrating a sound knowledge of the basics. In (c), a common mistake was to assume that distributed memory cannot be shared. In (d), the phenomenon of false sharing was well understood, though not all candidates illustrated their answers with the specific example that the question requested. In (e), it was pleasing to see most candidates identifying DMA as being relevant, even though the question did not explicitly refer to it. As with all essay-style questions, the best responses were characterized by sound editorial judgment, addressing the question that was actually asked rather than just offering a general discourse about DMA.

3. **Virtual memory systems**

(a) There are two main requirements that motivate the adoption of a virtual memory system. The first is the desire to be able to write programs without having to worry about the amount of physical memory installed in the computer. The second is the need for the CPU to execute multiple processes separately: each process should be unaware of, and protected from, the others. [10%]

(b) (i) The TLB is fully associative, the cache is direct-mapped. The cache block size is 4 bytes (one word). [10%]

(ii) On a TLB miss, the address translation is looked up in the page table instead. This is slower than a TLB look-up, since the page table resides in main memory. If the page exists in main memory, then the TLB entry can be updated from the page table (replacing another TLB entry, perhaps the least recently used one) and the translation retried. If the page resides on disk, then a page fault occurs. The operating system fetches the required page from disk, swapping it with another page (usually the least recently used one) in main memory. To save disk writes when a page hasn't changed, the page table and TLB include a dirty bit to show whether the page's contents differ from the data on disk. After the page has been loaded into main memory, the page table and TLB are updated with the new translation and the translation is retried. [15%]

(iii) There are a total of 2^{20} pages. Each page table entry contains a physical page number (20 bits) along with valid and dirty bits (2 bits). Rounding up to the nearest whole byte, each page table entry is 3 bytes long. So the total size of the page table is $2^{20} \times 3 = 3$ MiB. For a 64-bit virtual address, there would be 2^{52} virtual pages. Assuming (say) 8 GiB of physical memory, the physical page number would be 21 bits wide, resulting in page table entries of 3 bytes, so a page table would occupy up to $2^{52} \times 3 = 12$ PiB. This is clearly not acceptable, especially seeing as one page table is required for each running process. [15%]

(c) By including a process identifier in the **inverted page table** entries, we can make do with a single, global page table shared between all processes. The inverted page table would then contain all the information we need: one entry per physical page, showing which (if any) virtual page is mapped to it. The process identifier is necessary since different processes might use the same virtual addresses.

The storage requirements for the inverted page table are reasonable and depend on the amount of physical memory, not the amount of virtual memory. For example, assuming 8 GiB of physical memory and 4 KiB pages, there would be 2^{21} rows in the inverted page table. For a 64-bit virtual address, each row would need to hold a virtual page number (52 bits), a process identifier (maybe 16 bits), and valid and dirty bits (2 bits). Rounding up to the nearest whole byte, this gives 9 bytes per entry. So the total storage requirement would be $9 \times 2^{21} = 18$ MiB.

The speed of address translation would be, at first sight, rather slow, since the inverted page table is not indexed by the virtual page number. Naively, we could perform a linear search of the entire table, checking every stored process identifier/virtual page number to

see if it matches the one we are trying to translate. If we get to the end of the page table without finding it, we have a page fault. Considering the example above, that would be 2M memory accesses to identify a page fault, and on average 1M memory accesses for a successful translation. With more sophistication, we could employ some sort of hashing strategy to decide where in the inverted page table an entry should be stored. And besides, we should bear in mind that page table lookups are not required for the vast majority of translations, given the efficacy of the TLB.

For the **multi-level** approach, each table clearly has to reside in a continuous region of physical memory. Hence, a one-page table is a good idea since the operating system kernel can simply grab another page of memory when it needs to create a new table, with no wastage. For the current example of 4 KiB pages and assuming 4-byte page table entries (a reasonable guesstimate), each table can have no more than 2^{10} rows if it is to fit in a page. The lower 12 bits of the virtual address are the page offset, leaving 52 bits for the virtual page number. We need to split these 52 bits into chunks of at most 10 bits, with each chunk indexing a table in the hierarchy. We therefore need six levels.

The speed of translation is linear in the number of levels in the hierarchy, so perhaps a little slower than a hashed inverted page table, but again this is of little consequence since most address translations are provided by the TLB. The amount of memory storage required for the page tables depends not on the size of the physical or virtual address space, but on the amount of memory that the process is actually using.

Comparing the two schemes, the multi-level approach would normally win on storage requirements (unless all the system's physical memory is in use by active processes), while there is little to choose between the two in terms of look-up time, which is anyhow of little consequence given the TLB. Another win for the multi-level approach is in dealing with shared memory for parallel processing. With a multi-level page table, it is perfectly possible for multiple virtual pages (from different processes) to point to the same physical page. For the inverted page table, we would need to somehow extend the table to accommodate multiple virtual page numbers/process identifiers for shared physical page numbers. It is not surprising, therefore, that inverted page tables are used in very few architectures. [50%]

Assessors' remarks: This question tested the candidates' understanding of virtual memory systems, in particular TLBs and page tables. The more familiar topics in parts (a) and (b) were well handled by the vast majority of candidates. There was a greater spread of marks in (c), where candidates were asked to extrapolate their knowledge beyond bookwork to consider the pros and cons of inverted and multi-level page tables. There were some excellent responses addressing all the salient points, but also some more sketchy answers that revealed the limits of the candidates' understanding. The most common weakness was a failure to realise that the amount of storage required for the multi-level page tables depends not on the size of the physical or virtual address space, but on the amount of memory that the process is actually using.

Andrew Gee
May 2021