Engineering Tripos Part IIB FOURTH YEAR

Module 4F14: Computer Systems

Solutions to 2023 Tripos Paper

1. **Instruction set architectures and memory addressing**

(a) An accumulator instruction set architecture uses a special register, the accumulator, as an implicit operand in many instructions. For example, consider adding two numbers at memory locations `MEM1` and `MEM2`, storing the result in `MEM3`:

```
ldaa MEM1   # accumulator loaded with number at MEM1
adda MEM2   # add number at MEM2, result put back in accumulator
staa MEM3   # accumulator stored at MEM3
```

Note how the arithmetic instruction `adda` gets one of its operands from memory. In contrast, a general purpose register, load-store architecture does not have a special purpose accumulator but a large number of general purpose registers. Arithmetic operations can only operate on numbers in registers: only load and store instructions can access memory. The addition would look like this:

```
lw $8,MEM1($0)  # register $8 gets number at MEM1
lw $9,MEM2($0)  # register $9 gets number at MEM2
add $10,$8,$9   # register $10 gets register $8 plus register $9
sw $10,MEM3($0) # register $10 stored at MEM3
```

Note how there are no implicit operands. [20%]

(b) MIPS load/store and branch instructions are I-format, in which the address field is 16 bits wide. MIPS jump/jump-and-link instructions are J-format, in which the address field is 26 bits wide. Load/store address offsets are in bytes, whereas branch/jump targets are in words. This means that: the offset in a load/store can be no less than `0x8000` = -32768 bytes or more than `0x7FFF` = $+32767$ bytes; the branch target in a branch cannot be more than `0x8000` = 32768 instructions backwards or `0x7FFF` = 32767 instructions forwards; the jump target must have the same upper four bits as the current contents of the program counter, i.e. it must be in the same 256 MB memory block as the jump instruction.

Practically, most programs will fit into a 256 MB memory block, facilitating fast, single-instruction jumps/jump-and-links. Exceptionally, it might be necessary to replace a very distant jump with a jump-register, loading the register first with the correct 32-bit address: this would be slower but functional.

The branch target limit is rarely a problem, since conditional branches mostly arise from loops and `if` instructions, jumping forwards or backwards to a nearby instruction. In practice, almost all loops and `if` clauses span fewer than 2^{15} instructions.

The limited load/store address offset would appear to be the most problematic, giving access to only a small portion of the 4 GB address space. In MIPS, programs would normally occupy the first 256 MB of memory, with data storage starting at address `0x10000000`, which would be well out of range for a load/store offset. However, a MIPS convention offers a convenient workaround, with `$28` reserved as a *global pointer* containing the address `0x10008000`. Then, with single instructions of the form `lw $8,0xXXXX($28)`, it is possible to access rapidly the first 2^{16} data bytes in the range `0x10000000` to `0x100FFFF`, which is where a MIPS compiler might store frequently used global variables. [40%]

(c) Let us first consider the interplay between the program **A** and the library **B**. Since they were compiled separately, they might use the same addresses for data and instructions. So one job the linker needs to do is *relocation*. This involves shifting the addresses of the instructions and data in **B** so as not to clash with **A**, and then editing all the *absolute* memory references in **B** (i.e. the address fields in jumps, loads and stores, but not branches) to reflect the new, relocated addresses. When **A** was originally compiled, there would have been unresolved references to functions and data in **B**. The linker can now resolve these by inserting the post-relocation addresses from **B**. Object files (as produced by the partial, independent compilation of **A** and **B**) contain symbol tables and relocation information to make the linker's job easier.

The interplay between the executables **C** and **D** is different. Again, they might use the same addresses, but this time we want to protect them from each other rather than link them together. So this has nothing to do with the linker and everything to do with virtual memory. All the addresses we have been referring to up until now have been *virtual* addresses. The operating system translates these into *physical* addresses, ensuring that there are no conflicts. Virtual and physical memory are divided into chunks called *pages*: the translation is between a virtual page number and a physical page number. The translations are stored in a structure called a *page table*, which is indexed by the virtual page number and contains the physical page number. The page table is cached in a *translation lookaside buffer* to facilitate rapid translation almost all the time, with no need to access the page table in main memory. [40%]

Assessors' remarks: This question tested candidates' understanding of instruction set architectures and addressing. Most candidates answered (a) well, explaining clearly the differences between load/store and accumulator architectures. In (b), even though most candidates understood that the various MIPS address fields were somewhat restricted, they struggled to elucidate the practical implications of said restrictions. In (c), although many candidates correctly identified virtual memory as being key to the concurrent execution of **C** and **D**, few were able to discuss the role of the linker in combining **A** and **B**. Although this material was not explicitly lectured, the wording of the question prompted candidates to think about how references from **A** to **B** might be resolved, but few were able to run with this hint.

2. **Pipelined datapaths and branching**

(a) Branch hazards occur when the address of the next instruction is required (for instruc-

tion fetching) before an earlier conditional branch instruction has been evaluated. They can be resolved by any one of, or a combination of, (i) stalling the pipeline until the address is known, (ii) assuming the branch is not taken and then flushing the pipeline if it is, (iii) more sophisticated forms of dynamic branch prediction, followed by flushing if necessary, (iv) *delayed branches*, whereby the instruction following a branch is always executed irrespective of whether the branch is taken. [20%]

(b) The comparator is cascaded after the register file. If the comparator is slow, there is a danger that ID might become the rate-limiting stage of the pipeline and the clock speed would need to be reduced. However, a fully functional (and hence relatively slow) ALU is not needed to compare the registers. We just need to know whether they are equal or not. This could be done by a parallel, bit-wise XOR followed by an AND gate, with no need for any carries. Using this sort of fast comparator, in place of a full ALU, saves hardware and reduces the risk of the clock rate having to be reduced. [20%]

(c) Given the single branch delay slot, the instruction after the `beq` is always executed.

```
        slt $11,$8,$9    # set $11 to 1 if $8 < $9
        beq $11,$0,foo   # branch if $8 ≥ $9
        add $10,$9,$0    # copy $9 to $10 always
        add $10,$8,$0    # copy $8 to $10 only if $8 < $9
foo
```

Assuming data forwarding to resolve the `$11` hazard between `slt` and `beq`, this code takes three or four clock cycles, depending on whether `$8`<`$9` or not. [20%]

(d) With the new instructions, we can write code that always takes just three clock cycles.

```
slt $11,$8,$9    # set $11 to 1 if $8 < $9
movn $10,$8,$11  # copy $8 to $10 if $8 < $9
movz $10,$9,$11  # copy $9 to $10 if $8 ≥ $9
```

In this example, and again assuming data forwarding, we have reduced the expected instruction count from 3.5 to 3. More generally with pipelined datapaths, it is desirable to avoid conditional branches since these are susceptible to branch hazards and hence delays of one sort or another. [20%]

(e) `movz` could be mapped to an R-format instruction:

op	rs ($0)	rt ($11)	rd ($10)	shamt ($9)	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

We have used the `rt` and `shamt` fields to select the two source registers `$11` and `$9`, and `rd` to select the destination register `$10`. If we then select `$0` in `rs`, we can use existing elements of the datapath to add `rs` ($0) and `rt` ($11), with the ALU's Zero output determining whether to write `$9` back to `rd` (`$10`) or not. The required changes to Fig. 1 are:

- Register file to output three registers, with the `shamt` field selecting the new Read data 3 output.
- The ALU output no longer goes directly to MUX3. Instead it goes via a new multiplexor, MUX5, which selects between the ALU output and Read data 3.
- A new control signal Movz switches MUX5. Movz will be high for `movz` and low for all other instructions.
- The RegWrite control signal no longer goes directly to the register file. Instead it goes via a new multiplexor, MUX6, which selects between RegWrite and the ALU's Zero output. MUX6 is switched using the Movz control signal.
- All other control signals are the same for `movz` as for `add`. [20%]

Assessors' remarks: This question tested candidates' understanding of pipelined datapaths and hazards. In (a), almost every candidate was able to discuss branch hazards and their mitigation, though a few confused branch hazards with data hazards. In (b), the better responses acknowledged the possibility of the ID stage becoming the rate-limiting stage of the pipeline, and realised that a comparator could be designed using bit-wise, parallel operations with no carries. Responses to (c) were disappointing, with most candidates failing to make good use of the branch delay slot. In contrast, most candidates were able to use the new MIPS instructions correctly in (d) and could explain the advantages over the original code in (c). Responses to (e) were mixed in terms of clarity and detail, with the better answers identifying the key role played by `$0`.

3. **I/O and DMA**

(a) The move to serial point-to-point networks arises because buses cannot keep up with the bandwidth of today's I/O devices. The clock rate of a parallel bus is limited by noise, stray capacitance, crosstalk and clock skew. If we want a fast parallel bus, we would need to make it short and limit the number of devices connected to it. This conflicts with the basic requirement that an I/O bus should be long and support as many I/O devices as necessary.

These problems are avoided by doing away with the shared, parallel bus, and replacing it with a switched point-to-point network. To mitigate the effects of different timing skews on each bit, these networks are serial. But they can run very fast, since there are only two devices on each link — so less load, noise and stray capacitance. These new I/O networks also circumvent the need for bus mastering protocols. Data transfer is typically synchronous to an embedded clock using, for example, 8b/10b encoding.

Serial connections have the added advantage of requiring fewer wires. This means less clutter and hence better air flow/cooling inside computer cases. Two examples in recent PC hardware are the transition from parallel ATA to serial ATA for disk devices, and from PCI/AGP to PCI express for generic I/O devices and graphics cards. [25%]

(b) Polling, interrupt-driven I/O and direct memory access (DMA) are three different mechanisms for allowing the CPU to interact with I/O devices. Polling requires the least

hardware: the CPU periodically checks to see whether the device is ready to send or receive more data, and handles the data transaction if necessary. The polling frequency must be high enough to satisfy the device's maximum data transfer rate. This can be tremendously wasteful of CPU time, especially for devices which are mostly idle. Polling may be used for low bandwidth devices which can tolerate low frequency polling, like mice.

Interrupt-driven I/O requires extra signal lines to interrupt the CPU whenever an I/O device requires attention. The CPU must still be involved in every bus transaction, so may still be heavily loaded when the device is active. But, in contrast to polling, there is no CPU load when the device is idle. Interrupt driven I/O may be used for relatively low bandwidth devices which are mostly idle, like printers.

DMA is the most expensive technique in terms of hardware, requiring dedicated DMA controllers. But it is the only viable technique for high bandwidth devices like disks, which might otherwise fully occupy the CPU with bus transfers. 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. Care needs to be taken with cache coherency and the interaction with the virtual memory system, i.e. whether the DMA controller is supplied with virtual or physical addresses. [25%]

(c) Having `status` and part of `buffer` share a cache block is something that should be avoided at all costs. To see why, let's go through the DMA process step by step.

- The driver starts a DMA transfer into `buffer`. As part of this operation, the coherency protocol will invalidate the cache blocks containing `buffer` and also (because it shares a cache block with `buffer`) `status`. Let's refer to the cache block contents at this stage as `status_0` and `buffer_0`.
- During the DMA process, the driver periodically reads and writes `status`. Assuming a write-back cache, the first time this happens the block will be copied from main memory into the cache: let's say its contents are now `status_0` and `buffer_1` (since `buffer` is in the process of being updated by the DMA). The driver then updates `status` to `status_1`.
- At the end of the DMA process, the memory contents will be `status_0` and `buffer_2` (since there have been further DMA updates to `buffer`).

At this point, both copies of the troublesome cache block are incorrect. The correct contents should be `status_1` and `buffer_2`; in the cache we have `status_1` and `buffer_1`; and in main memory we have `status_0` and `buffer_2`. There is nothing that can be done to recover from this situation. If the processor accesses data in `buffer`, it will receive (from the cache) stale data from part way through the I/O process. If the cache block is written back to main memory, `buffer_2` will be incorrectly overwritten by `buffer_1`. If we invalidate the cache block again at the end of the DMA operation, we will lose the changes made to `status`.

One solution is to route all DMA activity through the cache, but this is expensive and wasteful of cache space. Alternatively, we could adopt a more sophisticated, snooping coherency protocol, like the ones used to facilitate inter-thread data sharing in SMPs. But this, again, is an expensive work-around to solve a problem that should not arise in the first place. Unlike data sharing in SMPs, which is necessary due to the nature of parallel processing, there is no reason why the DMA buffer should be touched by other running threads while the transfer is happening. The coherency protocol should therefore stipulate that DMA-sensitive buffers must not share cache blocks with other data. [50%]

Assessors' remarks: This question tested candidates' understanding of I/O and DMA. Most candidates demonstrated a good understanding of the principles in (a) and (b), though a few inexplicably discussed network-connected MIMD parallel processors. Part (c) asked candidates to dissect a particular, unfamiliar problem with DMA and device drivers. Coherent arguments were generously rewarded, even if they failed to appreciate every nuance of the specific situation. However, many candidates ignored the question and simply wrote down everything they knew about DMA, receiving little credit in return.

Andrew Gee
May 2023