Engineering Tripos Part IIB FOURTH YEAR

Module 4F14: Computer Systems

Solutions to 2025 Tripos Paper

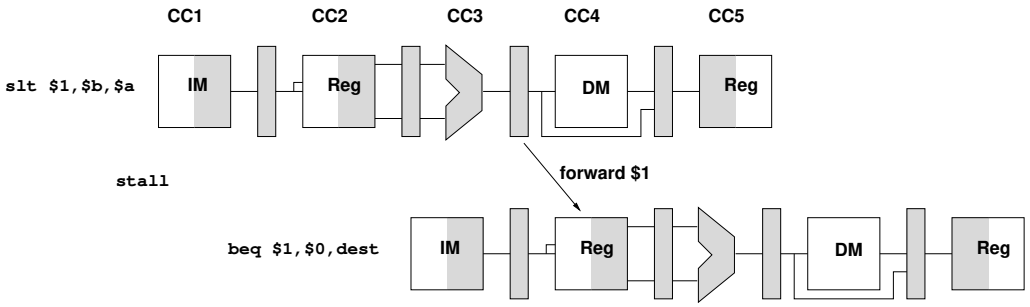
1. **Pipelined datapaths and branching**

(a) Branch hazards occur when the address of the next instruction is required (for instruction 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) By moving the branch decision from the MEM stage to the ID stage, there would be only one possibly incorrect instruction following the branch in the pipeline, instead of three. This would enable a delayed branch with one delay slot and no flushing or stalling whether the branch is taken or not; or, alternatively, an immediate branch with the need for just one stall or flush if the branch prediction is incorrect.

In terms of precautions, note that 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) Note the `$1` read-after-write data hazard between the `slt` and the `bne`. We could reduce the number of necessary stalls to just one if we forward the result of the `slt` from the EX/MEM pipeline register to the comparator input at clock cycle 4:



But, even with data forwarding, the branch could not be resolved any sooner than this.

A direct hardware implementation of `ble`, `bgt`, `bge` and `blt` would require a full ALU for subtraction and not the fast comparator in Fig. 2. To avoid ID becoming the rate-limiting step, the hardware implementation would likely resemble Fig. 1, with branch resolution at clock cycle 4. This suggests a rationale behind the design of the MIPS ISA, with only `beq` and `bne` implemented directly, since this allows fast branch resolution for these two instructions as in Fig. 2, with no detriment for the other conditional branches which are still resolved at clock cycle 4 through `slt`-type pseudo-instructions. Clearly, fast branch resolution is desirable, even if only for two types of branch, since it reduces the need for branch hazard mitigation. [30%]

(d) The CCR cannot be written until the result of the ALU or data transfer instruction is known, so no sooner than pipeline stage 3 for ALU instructions and stage 4 for `lw`. A subsequent branch instruction would wish to read the CCR, and hence resolve the branch, as early as possible, which would be immediately after instruction decoding at pipeline stage 2. However, this introduces the potential for read-after-write data hazards, which would need to be resolved in the usual way through hazard detection and stalling. Alternatively, we could simplify the hardware design by having the CCR written and read at the same pipeline stage (this would need to be stage 4, so as to accommodate `lw`), eliminating the possibility of data hazards but increasing the number of stalls required for branch hazards, since branch resolution would now be later.

In effect, CCR branching is not unlike the `slt`-type pseudo-instructions considered in (c), with one instruction setting a status bit and a subsequent instruction testing that bit to decide whether to branch or not. The big difference, however, is that the MIPS approach is explicit whereas the CCR approach is implicit, with CCR bits changed as a side-effect of almost all instructions. This has profound implications for dynamic pipeline scheduling, since any instruction scheduled in an otherwise unfilled slot has the potential to effect the CCR and hence break a subsequent branch instruction. In effect, the number of potential hazards that the scheduler must consider is increased dramatically, often for no purpose since most CCR bit updates are *not* subsequently tested by a later branch instruction.

Finally, whereas a MIPS `slt` can write to *any* register, there is only one physical CCR and hence the need for scheduling restrictions with superscalar pipelines, since only one instruction can write to the CCR at a time. [30%]

As an aside, it is telling that although ARM uses a CCR for branching, updating the CCR is *optional*, so the programmer/compiler can touch the CCR only when strictly necessary for a subsequent branch test. This is more or less the same as the MIPS `slt`-type pseudo-instructions, just dressed up differently.

Assessor's remarks: This question tested candidates' understanding of pipelines and hazards. The book work in (a) was answered well by the vast majority of candidates, though a handful confused data and branch hazards. Similarly, in (b), almost all candidates understood the purpose of the modification and the need to make the comparator fast. In (c), most candidates spotted the data hazard between the two instructions but not all suggested data forwarding as a way to reduce the number of stalls, making it rather difficult to justify

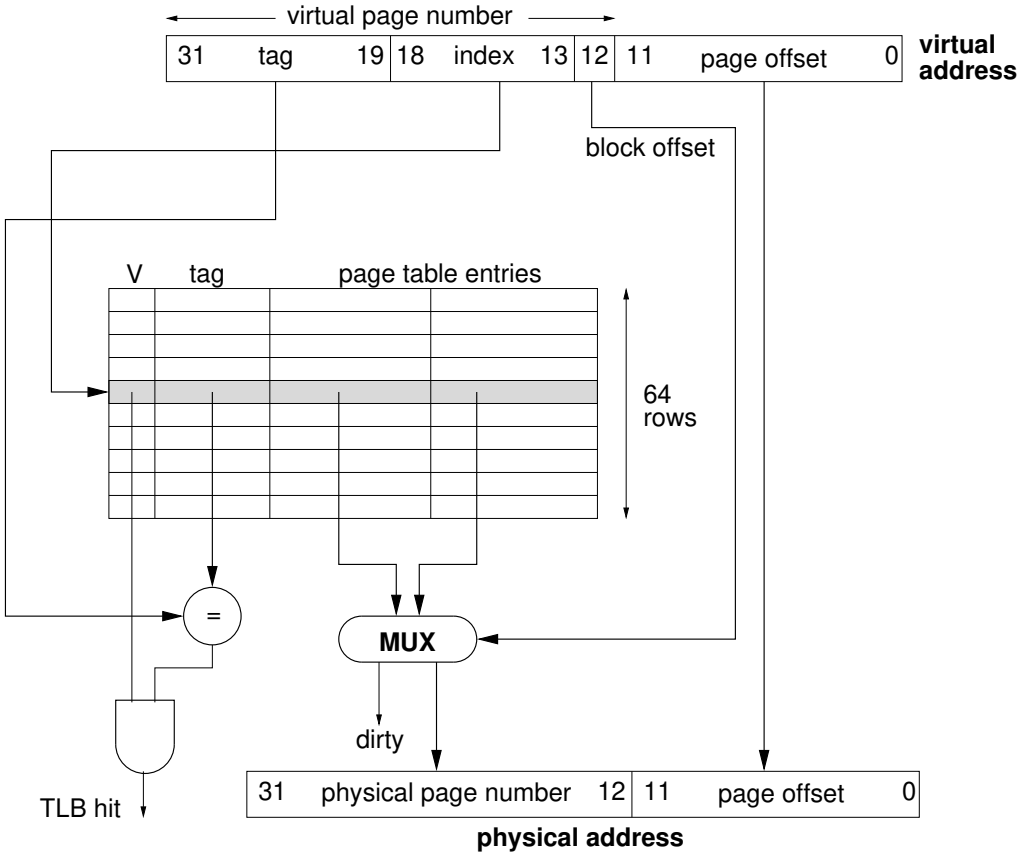
the use of pseudo-instructions. In (d), only a handful of students hit the nail on the head, but many others nevertheless received credit for sensible discussions that demonstrated some understanding of the key issues.

2. **Virtual memory, TLBs and caches**

(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.

However, the need to look up address translations in the page table (which is large and therefore stored in main memory) undermines the existence of the cache. So the CPU typically contains a small, fast cache called the translation lookaside buffer (TLB) which caches recently used page table entries. Locality of reference to the page table means that the TLB miss rate is typically very low. [25%]

(b) (i)



In the event of a TLB miss, if the page exists in MM, the TLB entry can be updated from the page table and the translation retried. Otherwise, a page fault is generated. [20%]

(ii) The process will repeat a large number of `lw $a, sw $b` instruction pairs, where `$a` iterates through array `A` and `$b` iterates through array `B`. The starting addresses for `A` and `B` in binary are $10000000\ldots0000$ and $10010000\ldots0000$, so `$a` and `$b` will differ only in bit 28 throughout. Bit 28 is in the TLB tag, not the index, so the indices into the TLB will be the same for each `lw`/`sw` pair. This is going to be catastrophic for performance: every address translation will result in a TLB miss and a main memory access to read the page table, regardless of whether the required words from `A` and `B` are in the data cache.

The problem is that the TLB is direct-mapped. TLBs are small and are therefore amenable to fully associative design, since there are only a small number of tags to check in parallel. LRU block replacement is difficult to implement in hardware, but even with random block replacement the situation would be improved considerably. [25%]

(c) Compared with our small TLB (just 64 rows in the example above), caches are much larger, even L1 caches at the top of the hierarchy. A 64 KiB L1 cache with a block size of four would have 4096 rows, so a fully associative design is going to be infeasible given our overriding desire for a short hit time. Hence, compared with TLBs, instruction/data caches are in general considerably larger with less associativity. L1 caches may even be direct-mapped.

Returning to the pattern of memory accesses in (b)(ii), it is important to understand that the data cache would normally work with physical addresses. Only bits 0–11 (the page offset) of the physical address are knowable, with bits 12–31 depending on which physical page the operating system has allocated. The L1 cache example above (64 KiB, four-word blocks) would use bits 4–15 for the index if direct-mapped. Address bits 0–11 are the same for `A` and `B`, but as long as bits 12–15 of the physical page numbers are not identical, they would not be competing for the same cache row. If we did not want to leave this to the operating system to get right, the compiler/programmer could guarantee cache-friendly behaviour by making sure that the page offsets of the two virtual addresses differ. For example, starting `A` and `B` at virtual addresses 0x80000000 and 0x90000800, respectively, would work. [30%]

Assessor's remarks: This question tested candidates' understanding of virtual memory, TLBs and caching. The book work in (a) was well answered by all candidates, as expected. In (b)(i), many candidates did well though others sketched generic cache configurations without explicitly showing how the addresses are divided into page offsets and page numbers. In (b)(ii), it was pleasing to see many candidates identifying the TLB conflicts and suggesting increased associativity as the solution. (c) was not well answered: while many candidates identified size and associativity as the key issues, very few had anything coherent to say about the pattern of cache accesses, since most missed the important detail that caches normally work with *physical* addresses. There were many responses that were simply lecture note dumps, hoping to stumble upon a salient point. Such responses received little credit.

3. ISAs and I/O

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

Note how there are no implicit operands and hence longer instructions. Finally, in a stack architecture all operations occur at the top of the stack, with only push and pop instructions accessing memory. The addition would look something like this:

```
push MEM1   # number at MEM1 pushed onto the stack
push MEM2   # number at MEM2 pushed onto the stack
add         # add numbers at the top of the stack, result
            # placed at the top of the stack
pop MEM3    # number at the top of the stack stored at MEM3
```

The accumulator instructions would require around three bytes each: one for the opcode and two for each address, giving a total of 9 bytes. The GPR instructions would be around four bytes each, one for the opcode, one for each register identifier and two for each address, giving a total of 16 bytes. Following similar reasoning, the stack instructions would be $3 + 3 + 1 + 3 = 10$ bytes.

At first sight, it would appear that load-store, GPR instructions have a disadvantage, in that they require more and/or longer instructions to be fetched from memory, with implications for execution time: more fetch-decode-execute cycles and/or more instruction cache misses. However, the benefits of the RISC approach generally outweigh the loss of instruction concision.

In particular, fixed length instructions are more amenable to pipelining, and the large number of general purpose registers (in contrast to a single accumulator) provides temporary

variable storage and hence reduces memory traffic for data. The load-store approach facilitates scheduling, since ALU operations are not susceptible to unpredictable data access latencies (cache misses/stalls). ALU operations can also be scheduled alongside data transfer instructions in superscalar pipelines, with no danger of data cache resource contention. With just a few instruction formats, limited operations and simple addressing modes, there is no need for complex datapath control (microcode). Some RISC architectures even overcome the code density handicap by offering two fixed length instruction formats, usually 32-bit and 16-bit. The different instruction encodings cannot be mixed freely but must be switched between explicitly. [50%]

(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 a dedicated DMA controller on the processor-memory bus. But it is the only viable technique for very high bandwidth devices, like graphics cards and disks, which might otherwise fully occupy the CPU with bus transfers. With DMA, the CPU hands control to the 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. [25%]

(c) DMA poses problems 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 are two potential problems here. When transferring data from the I/O device to memory, there may be a copy of this chunk of memory in the cache. If the processor reads from this chunk, it will get the old value (from the cache) and not the new value (as updated by the DMA controller). Similarly, when DMA is used to transfer data from memory to the I/O device, and the cache is write-back, the DMA controller may

transfer an old value from memory when there is a newer one in the cache. This is called the *stale data problem*.

There are three ways round this. One possibility is to route all DMA activity through the cache, though this is expensive and very wasteful of cache space, since the processor rarely needs to see the I/O data immediately and, in the meantime, useful data has been displaced from the cache. The second option is to have the operating system invalidate the cache 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 hardware mechanisms to selectively flush individual cache entries. [25%]

Assessor's remarks: The first part of this question asked candidates to investigate the code density of accumulator, stack and GPR load/store ISAs. It was well answered by many, though some candidates ignored the number of instructions, focusing exclusively on their length. When candidates were asked to explain why GPR load/store ISAs are nevertheless attractive, a coherent response would first need to explain why low code density is a problem: more execution cycles and/or more instruction cache misses. Instead, many candidates simply wrote down everything they could remember about GPR load/store ISAs, without addressing the question directly. The second part of the question was book work about I/O and was very well answered by the vast majority of candidates.

Andrew Gee
May 2025