Engineering Tripos Part IIB FOURTH YEAR

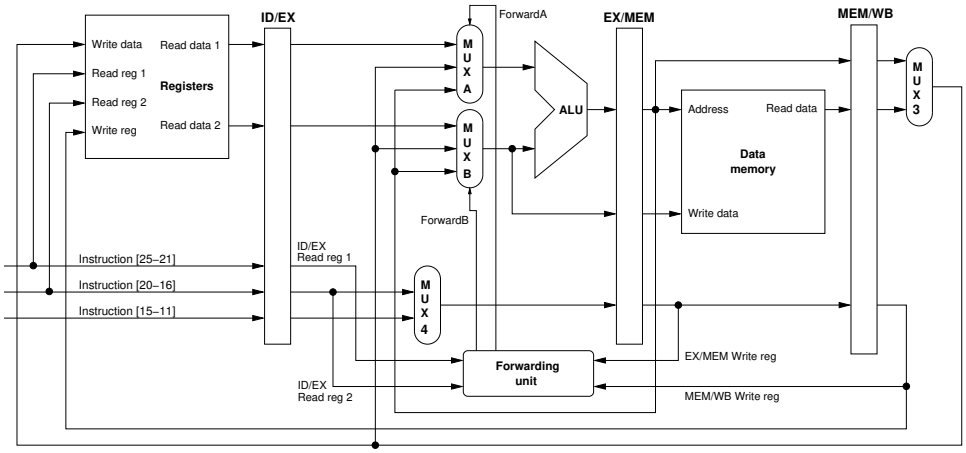
Module 4F14: Computer Systems

Solutions to 2018 Tripos Paper

1. **Datapaths and pipelining**

(a) Without pipelining, the longest route through the datapath (for a `lw`) requires $2+1+1+2+1=7$ ns, and this determines the clock speed. With pipelining, we could clock the datapath at the speed of the slowest pipe stage, which is 2 ns. Even though each instruction takes at least $2\times 5=10$ ns to clear the pipeline, asymptotically we hope to be completing one instruction every 2 ns, so a speed-up of $\times 3.5$ compared with the non-pipelined version of the datapath. In practice, there will be stalls due to data and branch hazards, which will reduce the speed-up. There will also be stalls due to cache misses, though the non-pipelined datapath is also susceptible to these. [20%]

(b)



The forwarding unit compares the registers which have just been read ("read reg 1" and "read reg 2") with any registers about to be written by the two downstream instructions. If they are the same, it sets MUXA and/or MUXB to ignore the value read from the register file and instead use the appropriate forwarded value. [20%]

(c) It is not possible to take full advantage of the superscalar resources without unrolling the loop. To see this, think about which instructions we might schedule at the same time as the `lw`, and in the two slots immediately afterwards. Not the `add` instruction, which depends on the `lw` result and, even with data forwarding, cannot follow it immediately. Not the `sw` instruction, which depends on the result of the `add` and so cannot be scheduled before it. Only the `addi` instruction can be moved forward, with the `sw` offset adjusted accordingly, but this would still leave two unused instruction slots in the superscalar pipeline.

Focusing on the body of the loop, and putting superscalar operation aside for now, we can remove the data hazards by unrolling as follows:

```
lw $8,Astart($9)    # $8 loaded with data at address $9+Astart
lw $12,Astart+4($9) # $12 loaded with data at address $9+Astart+4
add $8,$8,$10       # $8 loaded with $8+$10
add $12,$12,$10     # $12 loaded with $12+$10
sw $8,Astart($9)    # $8 stored at address $9+Astart
sw $12,Astart+4($9) # $12 stored at address $9+Astart+4
```

These six instructions process elements i and $i+1$ of the array. Since there are no dependencies *between* loop iterations, and no instruction sequencing constraints, with the superscalar hardware we could schedule a similar set of six instructions concurrently with those above, processing elements $i+2$ and $i+3$ of the array. This would take full advantage of the superscalar resources. The loop maintenance instructions (`addi` and `bne`) are dependent and would therefore need to be scheduled sequentially. There might be some further stalls depending on how branch hazards are resolved, but the effects will be small if the loop is unrolled enough times. [40%]

(d) A disadvantage of static scheduling is that it can only handle dependencies that are known at compile time. Dependencies that involve a memory reference may not be detectable. For example, reading `A[i]` after writing `A[j]` may or may not be hazardous, depending on the values of `i` and `j` at run time. Also, static scheduling assumes a particular pipeline: compiler-optimized code for pipeline A may not run efficiently on pipeline B. Static scheduling also implies compiler complexity and longer compilation times.

In contrast, dynamic scheduling does not suffer from these issues, but there is a significant increase in hardware complexity. It is not feasible for simple hardware to remove data dependencies altogether: rather, the idea is to process independent later instructions while waiting for stalls to be resolved. Thus, dynamically scheduled code may not always be as efficient as statically scheduled code. To get the best of both worlds, the two techniques are commonly used together. [20%]

Assessors' remarks: This question tested the candidates' knowledge of datapaths, pipelining and hazards. In (a), most candidates were able to calculate the maximum asymptotic speedup and to identify that hazards were the complicating factor. In (b), most candidates produced nice sketches showing the data forwarding unit and the extra multiplexors. In (c), most candidates identified the hazards and suggested how to overcome them by reordering/unrolling, but relatively few addressed properly how to schedule the instructions on *superscalar* hardware. In (d), most candidates came up with the obvious differences between static and dynamic pipeline scheduling (hardware vs. compiler complexity), but any deeper insight was rare.

2. **Page tables and virtual memory**

(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. [15%]

(b) Assuming none of the pages are initially present in physical memory, the first three pages (1 0 2) will generate page faults as they are fetched into physical memory. Next, there is another page fault, with page 3 replacing page 1, leaving pages 0 2 3 in physical memory. The following access to page 0 is a hit, but the next access to page 1 is a miss. There are therefore five page faults in total. [15%]

(c) $4\,\text{KB} = 2^{12}$ bytes per page. So the lower 12 bits of the virtual address are the page offset, and the upper 20 bits are the virtual page number. The page table requires one row per virtual page number, so the total memory requirement is $2^{20} \times 4\,\text{bytes} = 4\,\text{MB}$. One such page table *per process* is required. [15%]

(d) The page directory and each page table require $2^{10} \times 4\,\text{bytes} = 4\,\text{KB}$ storage. The first row of the page directory covers virtual page numbers 00000000000000000000 to 00000000001111111111, which is 0 to 1023 in decimal. Similarly, the last row of the page directory covers virtual pages 11111111110000000000 to 11111111111111111111, which is 1047552 to 1048575 in decimal. The process will therefore index only two rows of the page directory, requiring only two page tables. The total storage requirement is therefore $4 + 4 + 4 = 12\,\text{KB}$. [20%]

(e) To fit in a page, each table can have no more than 2^{10} rows. 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. A page 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. [20%]

(f) It is tempting to conclude that the translation time scales linearly with the number of levels in the hierarchy, and this is true up to a point: the time to retrieve a translation *on a TLB miss* will indeed scale linearly with the number of levels. However, TLB misses are rare: most translations are found in the TLB, and are therefore performed rapidly and independently of the number of page table levels. We would therefore expect the number of levels to have a negligible effect on the computational expense of virtual-to-physical address translation. [15%]

Assessors' remarks: This question tested the candidates' understanding of virtual memory and page tables. The book work in (a)–(c) was almost universally well answered. In (d)–(f), candidates were invited to consider the unfamiliar concept of hierarchical page tables. A pleasing number of candidates mastered the concept well, and were able to perform the necessary calculations in (d)–(e). Others, however, did not understand the hierarchical structure and produced correspondingly inaccurate calculations. In (f), although most candidates identified the linear complexity of traversing the hierarchy, only a couple realised that the TLB would render this largely irrelevant.

3. **Parallel processing and cache coherency**

(a)

SIMD, Single Instruction Stream Multiple Data Streams. A computer classification in Flynn's taxonomy of parallel processing machines. Multiple execution units respond to the same instruction at the same time, but operate on different data. Useful for vector processing, commonly used in graphics hardware.

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.

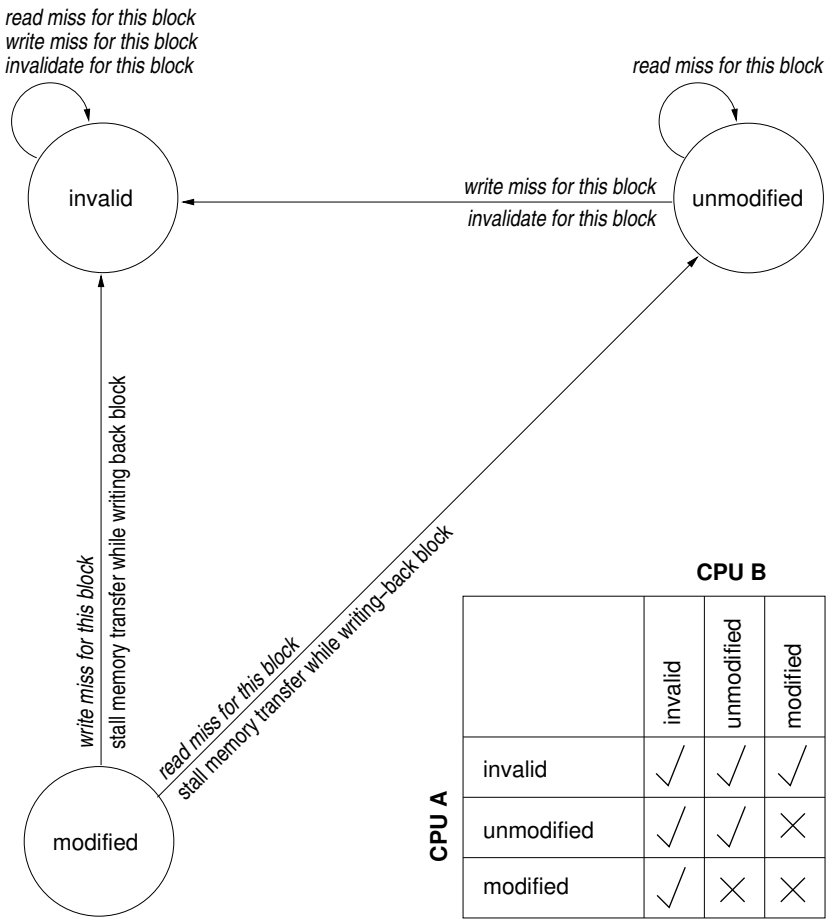
SMT, Simultaneous Multithreading. An operating mode for superscalar hardware that allows several processes to run concurrently, with instructions from distinct processes fetched in the same clock cycle. Compared with conventional process switching, the superscalar hardware spends less time idling since there are fewer dependencies between the running instructions. [30%]

(b) (i) and (ii)

See diagram overleaf. We have assumed that, on a read/write miss, it is the shared memory system that provides the block by default. So there is no need for a cache to provide blocks from the "unmodified" state. However, where a cache has a block in the modified state, it must write-back the block and stall the pending memory transfer as it does so.

A block that is "modified" in any one cache implies that it is "invalid" in all the others. To see this, observe how all transitions to the modified state involve broadcasting either a write miss or an invalidate signal, which will cause all other caches to make the block invalid. Furthermore, a "modified" block should never receive an "invalidate" signal, since such signals emanate from "unmodified" blocks, but we know all corresponding blocks will be "invalid". [70%]

Assessors' remarks: An unpopular question that started with some book work on parallel processing concepts, before seeing whether candidates could understand and develop the MSI cache coherency protocol in considerably more detail than was presented in the lectures. The book work was almost universally well answered. Attempts at developing the MSI protocol were more variable, with a bimodal mark distribution reflecting a group of candidates who clearly understood what they were doing, and another group who appeared to be mostly guessing.



CPU B

CPU A	invalid	unmodified	modified
invalid	✓	✓	✓
unmodified	✓	✓	✗
modified	✓	✗	✗

Andrew Gee
May 2018