Engineering Tripos Part IIB FOURTH YEAR

Module 4F14: Computer Systems

Solutions to 2019 Tripos Paper

1. **Datapaths and pipelining**

(a) A pipelined datapath features extra registers between the principal datapath stages. In each clock cycle, instructions advance through just one stage of the datapath, writing their interim results into the pipeline registers. In this way, several instructions can be in the pipeline at the same time, one at each stage. Pipelining therefore increases instruction throughput.

The term *hazard* is used to describe dependencies between instructions which disrupt the operation of a pipelined datapath. *Data hazards* occur when an instruction requires data before a previous instruction has written it to the register file. *Branch hazards* occur when the address of the next instruction is required (for instruction fetching) before an earlier conditional branch instruction has been evaluated. [20%]

(b) (i) Without data forwarding, there are read-after-write hazards between every adjacent pair of instructions apart from the `sw/addi`, each requiring three stalls. We also need to flush three instructions after the `bne`, apart from on the last iteration when the branch is not taken. Allowing for the initial `add` instruction (followed by three stalls) and also four cycles at the end for the pipeline to clear, the code requires $4+17(n-1)+14+4 = 17n+5$ cycles. [10%]

(ii) Forwarding cannot completely resolve the data hazard between the `lw` and the `add`, so there will have to be one stall here. The code now requires $1 + 9(n-1) + 6 + 4 = 9n + 2$ cycles. [10%]

(iii) We can avoid the stall between the `lw` and the `add` by unrolling the loop. This also reduces the number of branch flushes.

```
      add $9,$0,$0          # clear $9 to zero
Loop: 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
      lw $13,Astart+8($9)   # $13 loaded with data at address $9+Astart+8
      lw $14,Astart+12($9)  # $14 loaded with data at address $9+Astart+12
      add $8,$8,$10         # $8 loaded with $8+$10
      add $12,$12,$10       # $12 loaded with $12+$10
      add $13,$13,$10       # $13 loaded with $13+$10
      add $14,$14,$10       # $14 loaded with $14+$10
      sw $8,Astart($9)      # $8 stored at address $9+Astart
      sw $12,Astart+4($9)   # $12 stored at address $9+Astart+4
      sw $13,Astart+8($9)   # $13 stored at address $9+Astart+8
```

```
      sw $14,Astart+12($9) # $14 stored at address $9+Astart+12
      addi $9,$9,16        # $9 loaded with $9+16
      bne $9,$11,Loop      # jump back 13 instructions if $9≠$11
```

This code requires $1 + 17(n/4 - 1) + 14 + 4 = 17n/4 + 2$ cycles. [20%]

(iv) The instructions can be scheduled on the 2-way superscalar pipeline, with just one load/store at a time, as follows. Note how $9 is incremented by 16 at the start of the loop, so all the address offsets (apart from the first one) need decrementing by 16.

```
      add $9,$0,$0
Loop: lw $8,Astart($9)        addi $9,$9,16
      lw $12,Astart-12($9)
      lw $13,Astart-8($9)     add $8,$8,$10
      lw $14,Astart-4($9)     add $12,$12,$10
      sw $8,Astart-16($9)     add $13,$13,$10
      sw $12,Astart-12($9)    add $14,$14,$10
      sw $13,Astart-8($9)
      sw $14,Astart-4($9)     bne $9,$11,Loop
```

This code requires $1 + 11(n/4 - 1) + 8 + 4 = 11n/4 + 2$ cycles. [20%]

(c) The actual number of clock cycles might be higher because of instruction and data cache misses, requiring further pipeline stalls. Loop unrolling will most likely increase the number of instruction cache misses: even if the unrolled loop remains intact in the instruction cache, there are more instructions to fetch and therefore more misses during the first iteration. [20%]

Assessors' remarks: This question tested candidates' understanding of pipelined datapaths and hazards. It was well answered, with candidates demonstrating a sound grasp of the important principles. Weaknesses were generally restricted to the details (calculating, accurately, execution clock cycles) rather than substance, though nobody appreciated the potential impact of loop unrolling on instruction cache misses.

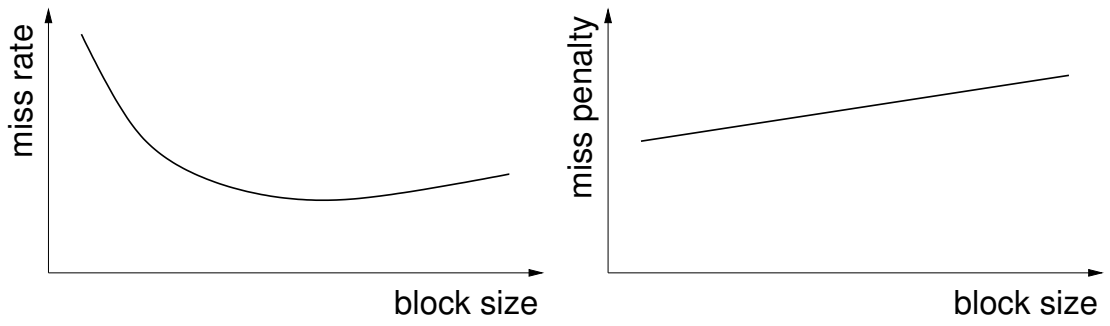
2. **Caches and locality of reference**

(a) Cache access is faster than main memory access because of its physical proximity to the CPU and its construction out of static RAM, as opposed to slower dynamic RAM. Data still needs to be fetched from main memory to the cache, but this overhead is easily amortized through temporal locality of reference (so a fetched item is likely to be accessed again soon, and this time it will be in the cache) and spatial locality of reference (so a fetched item's neighbours are likely to be accessed soon, so fetch *blocks* of data at a time, paying the main memory latency price just once). [10%]

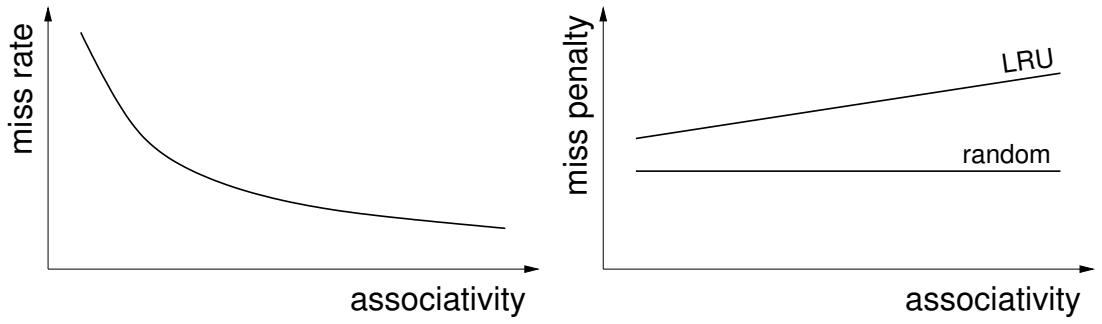
(b) Direct mapped caches have the lower hit time, since the index points to a unique block and no searching of the cache is required. However, since there is no choice of which

block to replace on a miss, we might replace blocks that are going to be referenced again soon: this will lead to a high miss rate. In a set-associative cache, the index points to a (typically small) set of blocks that must be searched, increasing the hit time. But there is some flexibility as to which block to replace on a miss. We could, for example, consider temporal locality of reference and replace the least recently used (LRU) block, thereby reducing the miss rate. [15%]

(c)



Spatial locality of reference means that the miss rate generally decreases with block size, though with very large blocks the miss rate may eventually increase (too many cache conflicts). The miss penalty increases with block size, since more words need to be transferred from main memory.



The miss rate decreases with increased associativity, since block replacement strategies like LRU can be used to replace blocks which are unlikely to be needed again soon. With LRU, the miss penalty may increase with more associativity, since the LRU algorithm needs to check more access times to decide which block to replace. For random replacement, the miss penalty is likely to be independent of associativity. [25%]

(d) (i) The data cache can store $1024/4 = 256$ `ints`. Since the matrix elements are all `ints` it follows that the cache can store 256 matrix elements at a time. [5%]

(ii) The code segment in Fig. 2 has very poor temporal locality of reference. Even though the elements of `b` and `c` are each referenced 1000 times, successive references to the same

element are not close together. The inner loop in `k` references 1000 distinct elements of `b` and 1000 distinct elements of `c` before there are any repeat references. These 2000 references will fill and refill the LRU data cache approximately 8 times over, with no cache hits. It follows that every reference to a matrix element in Fig. 2 will miss, apart from the repeat references to `a[i][j]`. Since there are 10^9 references to elements of `b` and `c`, and 10^6 references to unique elements of `a`, there will be approximately 2×10^9 cache misses. [20%]

(iii) The code segment in Fig. 3 has much better temporal locality of reference. The inner loops in `j` and `k` reference 100 elements of `c` and 10 elements each of `a` and `b`. All these elements will comfortably fit in the data cache. The next iteration of the `i` loop references the same 100 elements of `c` and 10 new elements each of `a` and `b`. Since the 100 elements of `c` are already in the cache, and they are all used each time round the `i` loop, there will be no further cache misses for `c` until the next iteration of the `kk` loop. So, for the inner three loops (in `i`, `j` and `k`) there will be a total of $100 + 2 \times 10 \times 1000 = 20100$ cache misses. The three inner loops are enclosed in two nested loops in `kk` and `jj`, each of which iterates 100 times. So the total number of data cache misses will be $100 \times 100 \times 20100 \approx 2 \times 10^8$, an improvement by a factor of 10 over the code in Fig. 2. [25%]

Assessors' remarks: This question tested candidates' understanding of caches and locality of reference. Candidates demonstrated an excellent level of understanding of the essential principles in parts (a)–(c). Analysis of the two matrix multiplication algorithms in (d) was somewhat variable. Even though most identified and evidently understood the superior locality of reference afforded by blocking, not many produced accurate estimates of the numbers of cache misses.

3. **Adders**

(a) Amdahl's Law says that it is of paramount importance to optimise the speed of those components that are most frequently used in a computer system ("make the common case fast"). For most instruction set architectures, ALUs are used at least once in the execution of just about every instruction. Faster ALUs would therefore have a significant impact on the speed of the overall system. [10%]

(b) Carry lookahead can be used to determine the carry inputs to each full adder without using ripple carry. For each bit i of the adder, we define two signals, generate gi and propagate pi. Bit i generates a carry if the two bits it is adding are both 1, and propagates a carry if either of the two bits it is adding is 1:

$$gi = ai.bi \qquad\qquad pi = ai + bi$$

c1, the carry into bit 1, will be 1 if either bit 0 generates a carry or c0 is 1 and bit 0 propagates a carry:

$$c1 = g0 + p0.c0$$

Likewise for c2, c3 and c4:

c2 = g1 + p1.g0 + p1.p0.c0
c3 = g2 + p2.g1 + p2.p1.g0 + p2.p1.p0.c0
c4 = g3 + p3.g2 + p3.p2.g1 + p3.p2.p1.g0 + p3.p2.p1.p0.c0

These expressions show how the carry-in signals can be obtained without waiting for them to ripple through a 4-bit adder. [20%]

(c)(i)

Type 1		Type 2		Type 2	
0100	0100	0100	0100	0110	0110
+ 1011	+ 1011	+ 1001	+ 1001	+ 1011	+ 1011
+ 0	+ 1	+ 0	+ 1	+ 0	+ 1
=01111	=10000	=01101	=01110	=10001	=10010

Carry-in (c0) and carry-out (cm) signals are highlighted in yellow.

The pattern that emerges is that in Type 1 additions, ai $\oplus$ bi = 1 for all i, and so the carry-in c0 is propagated through to the carry-out cm. In Type 2 additions, there is either a pair of bits that are both zero, killing any carry that was propagating from the right, or a pair of bits that are both one, generating a carry irrespective of any carry that was propagating from the right: either way, the carry-out cm is independent of the carry-in c0. The logic for the CL block is therefore Type 1 = (a0 $\oplus$ b0) . (a1 $\oplus$ b1) . . . (a(m-1) $\oplus$ b(m-1)) . [20%]

(ii) Let us denote the carry-in signals to each block as C0, C1 . . . C(k-1).

The latency of each block, including the multiplexor, is $(m+1)T$. After this amount of time, all of the m-bit adders would have produced their sums and carries, though only the first block correctly, since the others have spurious Ci signals initially. But C1 is correct at time $(m+1)T$.

If the second block is Type 2, C2 is already correct since it is independent of C1. If the second block is Type 1, C2 will be correct after a further delay T, the multiplexor latency. So C2 is correct at time $(m+2)T$.

Continuing this line of argument, we deduce that C(k-1) is correct at time $(m+k-1)T$. Allowing a further $(m+1)T$ for block k to produce its sums and carry, we find that the total latency is $(2m+k)T$. Note that block k will be the last to stabilize.

Now differentiate to find the optimal value of k:

$$t = (2m+k)T = \left(2m + \frac{n}{m}\right)T \Rightarrow \frac{dt}{dm} = \left(2 - \frac{n}{m^2}\right)T \Rightarrow \frac{dt}{dm} = 0 \text{ when } m = \sqrt{\frac{n}{2}} \quad [40\%]$$

(iii) In terms of asymptotic time/space complexities, block-carry-skip is $\mathcal{O}(n)$/$\mathcal{O}(n)$ while carry-lookahead is $\mathcal{O}(\log n)$/$\mathcal{O}(n \log n)$. For large n, therefore, carry-lookahead is likely to be considerably faster, albeit with a space penalty. For smaller n, block-carry-skip is roughly m times faster than ripple-carry, with a very small space overhead, and therefore worthy of consideration. Further speed-ups are possible using variable-size blocks. [10%]

Assessors' remarks: A relatively unpopular question on ALUs and adders. Candidates demonstrated an excellent understanding of carry lookahead adders in (b), but analysis of the unfamilar block-carry-skip adder in (c) was less successful. Although most candidates appreciated the different significance of the carry-in signal for type 1/2 additions, and correctly identified the logic required to distinguish between the two types, only two candidates calculated (and optimized) the adder's latency correctly.

Andrew Gee
May 2019