Engineering Tripos Part IIB FOURTH YEAR

Module 4F14: Computer Systems

Solutions to 2022 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, though the speed-up may be limited by data and branch *hazards* caused by dependencies between instructions. However, the deleterious effects of hazards can be mitigated by a number of effective strategies including data forwarding, branch prediction and delayed branches.

Suppose there are p pipe stages and n instructions to execute. Without pipelining, each instruction takes $(p-1+k)T$, so n instructions take $n(p-1+k)T$. With pipelining, all the stages need to run at the speed of the slowest stage, so we need to distinguish between the cases $k \leq 1$ and $k > 1$. For $k \leq 1$, the slowest stage requires T. The first instruction takes pT to execute, with each subsequent instruction adding a further T. The time to execute n instructions is therefore $pT + (n-1)T = (n+p-1)T$. For $k > 1$, the slowest stage requires kT. The first instruction takes pkT to execute, with each subsequent instruction adding a further kT. The time to execute n instructions is therefore $pkT + (n-1)kT = (n+p-1)kT$.

The speed-up is the time without pipelining divided by the time with pipelining:

$$\text{speed-up} = \begin{cases} \dfrac{n(p-1+k)T}{(n+p-1)T} \to p-1+k \text{ as } n \to \infty, & k \leq 1 \\ \dfrac{n(p-1+k)T}{(n+p-1)kT} \to 1 + \dfrac{(p-1)}{k} \text{ as } n \to \infty, & k > 1 \end{cases}$$

The maximum speed-up of p is achieved when $k = 1$. We have assumed no pipeline stalls. [40%]

(b) Without compiler optimizations, it is reasonable to assume that the C++ will be translated “as is” to machine code. For example, each loop in Code A will include a load for `x[i]`, an add to accumulate the sum, another add to increment `i` and a conditional branch depending on whether `i` is less than `n`. There will be many branches, each potentially taking several clock cycles if the branch hazards are not resolved efficiently.

In comparison, Code B has half the number of branches and `i` increments, which is a plausible explanation as to why it runs faster. On the other hand, the code is a little less compact, so we might expect more instruction cache misses, but this is evidently a secondary consideration compared with the benefits of the two-fold loop unrolling.

Finally, Code C exposes more instruction level parallelism by breaking the dependence between the two additions inside the loop. Whereas the two additions in Code B cannot run concurrently, they can in Code C. We are told that the execution times are for a modern processor, which we can therefore assume to be superscalar. The most likely explanation for Code C running faster is therefore the concurrent scheduling of the two additions on the superscalar pipeline. [30%]

(c) The point here is that the last two instructions are fundamentally independent of the first three, but the repeated use of $8 introduces a write-after-read dependency between the second add (write) and the preceding two instructions (read), which impedes out-of-order execution. This fake dependency can be eliminated by *register renaming*, which makes the code more amenable to out-of-order execution and, consequently, superscalar pipelines.

In this example, by renaming $8 to $13 in the last two instructions, the code can be scheduled on a 2-way superscalar pipeline as follows:

```
add $8,$9,$10       add $13,$11,$12
addi $8,$8,4        sw $13,B($0)
sw $8,A($0)
```

Note that there will not be any resource conflicts since there is only one load/store operation scheduled at a time.

Register renaming can be performed by the compiler, though such an approach would be limited to the set of architectural registers specified in the ISA. One can imagine how a processor might perform register renaming in hardware, by maintaining a dynamic map of the architectural registers to a much larger set of physical registers. [30%]

What follows is not expected of candidates, but is provided for the benefit of future generations of students who might be interested to learn more about hardware register renaming.

In a typical implementation, a new mapping is created from an architectural register to a physical register for every write operation, and later freed when the contents of the physical register are no longer required.

Returning to the present example, and assuming the following initial mapping from architectural to physical registers

```
$8->p11  $9->p12   $10->p13  $11->p14  $12->p15
```

with physical registers 16–200 free, the registers could be renamed dynamically, in hardware, as follows:

```
instructions        map table               renamed instructions
                    $8  $9  $10 $11 $12
                    p11 p12 p13 p14 p15
add $8,$9,$10       p16 p12 p13 p14 p15     add p16,p12,p13
```

```
addi $8,$8,4         p17 p12 p13 p14 p15     addi p17,p16,4
sw $8,A($0)          p17 p12 p13 p14 p15     sw p17,A(0)
add $8,$11,$12       p18 p12 p13 p14 p15     add p18,p14,p15
sw $8,B($0)          p18 p12 p13 p14 p15     sw p18,B(0)
```

Note how the renamed instructions correctly preserve read-after-write dependencies, but destroy write-after-read dependencies, making out-of-order execution more viable. There are, of course, many subtleties (e.g. when to free a physical register, how the `addi` instruction uses the old $\$8$ mapping for its source operand but the new mapping for its destination operand), but hardware register renaming is one of the most elegant concepts in computer architecture and is used in all modern processors. See also *Tomasulo's algorithm*.

Assessors' remarks: This question tested candidates' understanding of pipelined datapaths. In (a), most candidates offered a good summary of what is meant by a pipelined datapath, and most concluded correctly that all stages should have the same latency for optimal speed-up, though algebraic rigour was often lacking, with many not realising that different analyses were required for $k \leq 1$ and $k > 1$. In (b), a significant minority of candidates overlooked the simple fact that Code B has fewer branches and increments than Code A, instead arguing unconvincingly that the performance difference is due to more subtle issues involving caching and data hazards. When discussing Code C, solutions which failed to mention the word "superscalar" gained little credit. Part (c) asked candidates to extrapolate beyond the syllabus to examine the benefits of register renaming. It was pleasing to see most candidates understand how renaming allowed the instructions to be scheduled efficiently on the superscalar pipeline, and many speculated intelligently about the feasibility of implementing register renaming in hardware.

2. **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.

For a typical MIPS datapath without pipelining, the longest path is for the `lw` instruction, which needs to access sequentially the instruction cache, the register file, the ALU, the data cache and then the register file again. Assuming the register file has a latency of T and the ALU and caches have a latency of $2T$, the clock period could not be shorter than $8T$. If the ALU were half as fast, this would increase to $10T$, which is 25% slower.

With pipelining, and assuming the same latencies as above, the ALU (along with the caches) are the rate-limiting factor for the entire pipeline. So if the ALU were half as fast, the pipeline would also run half as fast, i.e. 100% slower. [20%]

(b) Suppose we are adding the two 64-bit binary numbers a63...a0 and b63...b0. Define two terms, generate (gi) and propagate (pi): gi = ai.bi, pi = ai + bi. We can use the generate

and propagate signals to design a simple, fast 4-bit adder:
c1 = g0 + p0.c0
...
c4 = g3 + p3.g2 + p3.p2.g1 + p3.p2.p1.g0 + p3.p2.p1.p0.c0
Four of these 4-bit adders can be connected using a higher level of carry-lookahead:
P0 = p3.p2.p1.p0
...
P3 = p15.p14.p13.p12
and
G0 = g3 + p3.g2 + p3.p2.g1 + p3.p2.p1.g0
...
G3 = g15 + p15.g14 + p15.p14.g13 + p15.p14.p13.g12
and
C1 = G0 + P0.c0
...
C4 = G3 + P3.G2 + P3.P2.G1 + P3.P2.P1.G0 + P3.P2.P1.P0.c0
Four of these 16-bit adders can be connected together, using carry lookahead, to make a 64-bit adder.
$P_{16}0$ = P3.P2.P1.P0
...
$P_{16}3$ = P15.P14.P13.P12
and
$G_{16}0$ = G3 + P3.G2 + P3.P2.G1 + P3.P2.P1.G0
...
$G_{16}3$ = G15 + P15.G14 + P15.P14.G13 + P15.P14.P13.G12
and
$C_{16}1$ = $G_{16}0$ + $P_{16}0$.c0
...
$C_{16}4$ = $G_{16}3$ + $P_{16}3$.$G_{16}2$ + $P_{16}3$.$P_{16}2$.$G_{16}1$ + $P_{16}3$.$P_{16}2$.$P_{16}1$.$G_{16}0$ + $P_{16}3$.$P_{16}2$.$P_{16}1$.$P_{16}0$.c0
Note that the carry signals Ci at the 4-bit level cannot be generated until the 16-bit carries are known. Likewise, the carry signals ci at the 1-bit level depend on the 4-bit carry signals.

Gate delay	Signals available	Generated from
0	ai,bi,c0	—
1	gi,pi	ai,bi
3	Pi,Gi	pi,gi
5	$P_{16}i$,$G_{16}i$	Pi,Gi
7	$C_{16}i$	$P_{16}i$,$G_{16}i$,c0
9	Ci	Pi,Gi,$C_{16}i$
11	ci	gi,pi,Ci
13	Sums	ai,bi,ci

So 13 gate delays are required to produce the sums and the carry-out. [30%]

(c)(i) The longest sum-of-product expression is for c63, i.e. the expression given in the question. This requires AND gates with up to 65 inputs and a 65-input OR gate, both of which could be constructed from (mostly) 4-input gates using the three-level hierarchy described in the question. So the 65-input AND would take three gate delays, followed by a further three gate delays for the 65-input OR. All of the other carry-in signals are no more complex than this, so we could calculate them all in six gate delays. The sums would come two gate delays later, add an extra one gate delay to calculate the propagate and generate signals in the first place, and we arrive at nine gate delays in total.

In terms of space, the single level design requires 65 gates to calculate c64 (counting just the terms in the sum-of-product expression, and neglecting for now how each large gate might be constructed from several smaller gates), 64 gates for c63, and so on down to 2 gates for c1, making 2144 gates. Of these, the largest gates (~ 64 inputs) would require 21 4-input gates in a three-level hierachy, the smallest would require only a single gate, so we can estimate 11 gates for each of the 2144, making 23584. Each full adder requires nine gates, and the propagate and generate signals require one gate each, making a total of $23584 + 704 = 24288$ gates. [20%]

(ii) For the multi-level design, if we use carry-lookahead to predict m carry-in signals at each level of the hierarchy ($m = 4$ in (b)), then we will need $\log_m n$ levels and gates with a fan-in of $\sim m$. Since each level takes a constant amount of time to operate, the asymptotic time requirement is clearly $O(\log_m n)$. In any straightforward silicon layout, the asymptotic space requirement is $O(n \log_m n)$.

For the single-level design, if we build high fan-in gates using hierarchies of m-input gates ($m = 4$ in (i)), then we will need $\log_m n$ levels, again giving an asymptotic time requirement of $O(\log_m n)$. The gate count, however, is much higher, since we require $1 + 2 + 3 + 4 \ldots + n = n(n+1)/2$ large gates to calculate the carry-in signals, $O(\log_m n)$ m-input gates to calculate each large gate, giving $O(n^2 \log_m n)$ overall. [20%]

(iii) Clearly, designs with high space requirements are less attractive since they require a larger silicon die. But we are not talking about a serious issue here for 64-bit adders. More pertinently, a higher gate count implies higher power dissipation, which is a significant issue for a critical component like an ALU that is used during the execution of every instruction. [10%]

Assessors' remarks: This question tested candidates' understanding of the role ALUs play in CPU datapaths, and how carry-lookahead adders are designed and implemented. Answers to (a) and (b) were generally good, with the vast majority of candidates coming to the right conclusions about the ALU's impact in pipelined and non-pipelined datapaths, and knowing how to design multi-level carry-lookahead adders. Responses to the unfamiliar part (c) were also good, with most candidates realising that the single-level and multi-level schemes have similar time but different space requirements. The better answers were supported by careful and accurate quantitative analyses, while the poorer ones failed to discriminate between the specific estimate required in (i) and the asymptotic analysis required in (ii). The only disappointment was how few candidates mentioned the increased power demands of high gate count designs in (iii).

3. **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) (i) The cache capacity is $4 \times 4 \times 65536 = 1048576 = 1\,\text{MiB}$. The index is given by

$$\left\lfloor \frac{\text{byte address}}{\text{bytes per block}} \right\rfloor \text{ modulo (blocks in cache)} = \left\lfloor \frac{\text{0xA0973C8E}}{\text{0x10}} \right\rfloor \text{ modulo 0x10000}$$
$$= \text{0xA0973C8 modulo 0x10000} = \text{0x73C8}$$

This is the penultimate byte in the block, so the word offset is 3 and the byte offset is 2. The tag comprises the remaining upper bits of the address, in this case 0xA09.

(ii) The cache capacity is $4 \times 4 \times 4 \times 4096 = 262144 = 256\,\text{KiB}$. The index is given by

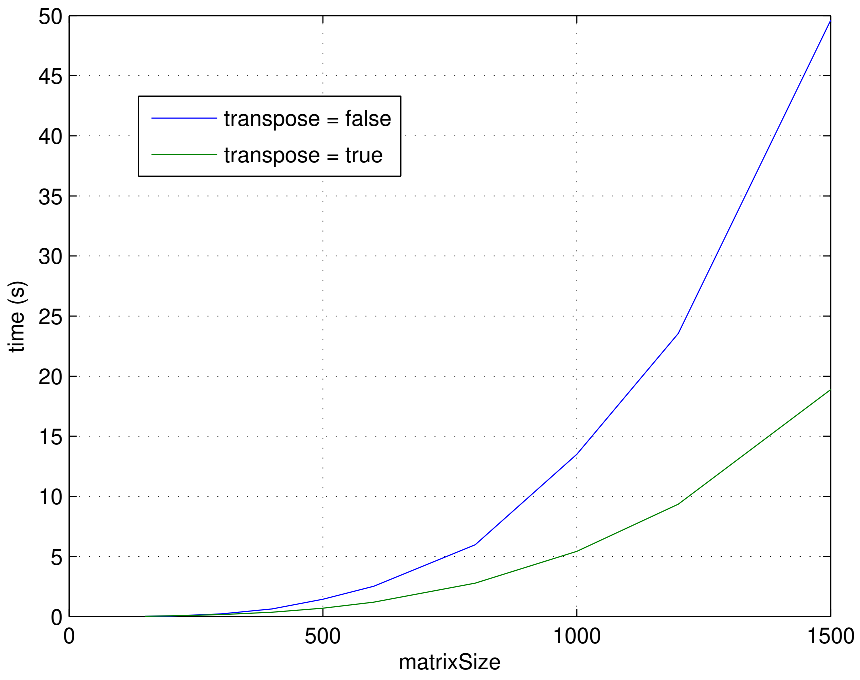
$$\left\lfloor \frac{\text{byte address}}{\text{bytes per block}} \right\rfloor \text{ modulo (sets in cache)} = \left\lfloor \frac{\text{0xA0973C8E}}{\text{0x10}} \right\rfloor \text{ modulo 0x1000}$$
$$= \text{0xA0973C8 modulo 0x1000} = \text{0x3C8}$$

This is the penultimate byte in the block, so the word offset is 3 and the byte offset is 2. The tag comprises the remaining upper bits of the address, in this case 0xA097. The byte may reside in any of the four blocks in set 0x3C8.

(iii) The cache capacity is $4 \times 4 \times 32768 = 524288 = 512\,\text{KiB}$. The byte may reside in any of the cache blocks. The tag would be the entire block address, which is 0xA0973C8 as before. This is the penultimate byte in the block, so the word offset is 3 and the byte offset is 2. [30%]

(c) Candidates are expected to deduce the curves' shapes themselves, but for interest on the next page are some actual results from an old 3.2 GHz Pentium 4.

There is the expected cubic dependency on `matrixSize`, but with a far lower cost per operation when `transpose = true`. This is because the matrices are evidently stored in memory row by row, so the elements of `b` and `ct` are accessed sequentially inside the inner loop, with excellent spatial locality of reference. The same cannot be said of `c`: cache misses account for the relatively poor performance when `transpose = false`. There is a small price to pay in taking the transpose: while it is hard to see on the graph, the curves do in fact cross at around `matrixSize = 225`.



[25%]

(d) The clock cycle time is $1/(2\,\text{GHz}) = 0.5$ ns. So the cost of accessing main memory is 100 clock cycles. The effective CPI with a single level cache is thus:

$$\begin{aligned}\text{effective CPI} &= \text{baseline CPI} + \text{memory stall cycles per instruction}\\ &= 1.0 + 100 \times 5\% = 6.0\end{aligned}$$

The cost of accessing the secondary cache is 10 clock cycles. The effective CPI with a two-level cache is thus:

$$\begin{aligned}\text{effective CPI} &= \text{baseline CPI} + \text{secondary cache stall cycles per instruction}\\ &\quad + \text{memory stall cycles per instruction}\\ &= 1.0 + 10 \times 5\% + 100 \times 50\% \times 5\% = 4.0\end{aligned}$$

The speedup is therefore $6.0/4.0 = 1.5$ times. We have assumed that the main memory access time, in particular the component associated with cache miss handling, remains the same irrespective of which cache it is servicing[1]. [25%]

(e) The primary cache filters accesses to the secondary cache, especially those with good spatial and temporal locality of reference. Hence, accesses to the secondary cache tend to have poor locality of reference, resulting in a high miss rate at this level. [10%]

[1]If the secondary cache has a larger block size than the primary cache, one might suppose that its miss penalty would be larger. However, there are schemes like *early restart* and *critical word first* that allow the CPU to restart before the entire block has been transferred from main memory to the cache.

Assessors' remarks: This question tested candidates' understanding of caches. In (a), almost all candidates demonstrated a sound understanding of the motivation for including a cache between the CPU and main memory. While there were many good answers to (b), a significant number of candidates could not calculate cache sizes, indices and tags, which was surprising given that a similar question featured on the examples paper. In (c), while most candidates realised how transposing the matrix improved spatial locality of reference, there were several who argued that this would somehow improve the $O(n^3)$ complexity of matrix-matrix multiplication! Answers to (d), estimating the performance improvement possible with a secondary cache, were satisfactory but not always entirely accurate. In (e), a pleasingly large number of candidates realised that the primary cache would filter accesses with good locality of reference, leading to a high miss rate in the secondary cache.

Andrew Gee
May 2022