Engineering Tripos Part IIB FOURTH YEAR

Module 4F14: Computer Systems

Solutions to 2024 Tripos Paper

1. **Instruction set architectures and adders**

   (a) Fixed-width 32-bit RISC instructions do not exhibit good code density. The MIPS `add` instruction is a good case in point: the 5-bit `shamt` field is unused, and we should not need to waste 12 bits on `op` and `funct` to specify `add`, since there are fewer than 2^{12} MIPS instructions! If we could reduce the memory requirements of the instruction stream, we might expect fewer instruction cache misses and hence fewer main memory accesses, decreasing both execution time and power consumption, the latter being especially important for mobile devices.

   Low code density is the price paid for the convenience of regular, predictable, fixed-width 32-bit instructions. CISC ISAs have better code density at the expense of variable length instructions, which might take several clock cycles to fetch.

   In contrast, with RISC 16-bit compressed instructions, no fetch capacity is wasted. In fact, two of them can be fetched at once. However, there are some obvious downsides too. Comparing the two `add` instructions, we see only 3-bit fields in MIPS16e for the register identifiers, allowing access to only eight of the 32 general purpose registers. Immediate operands, typically 16-bit in regular MIPS, would also need to be shorter (they are typically only 8 bits in MIPS16e). MIPS16e instructions might save further space by using the same register as both a source and destination operand. This all amounts to a smaller number of less powerful, less flexible instructions in MIPS16e than in MIPS. Any advantages of the compressed instruction stream might be immediately lost if, for example, we needed several compressed instructions to replace a regular 32-bit one.

   However, if the MIPS implementation allows cost-free switching between regular 32-bit MIPS and MIPS16e modes (as many of them do), it may be beneficial to code specific, amenable functions in MIPS16e, with the rest of the program in 32-bit MIPS. [50%]

   (b) A 16-bit ripple-carry adder is formed by daisy-chaining 16 full adders, with the carry-out of adder k connected to the carry-in of adder $k + 1$. Asymptotic time and space requirements are both clearly $\mathcal{O}(n)$.

   A 16-bit carry-select adder is formed of paired blocks of ripple-carry adders. One adder in a pair assumes the carry-in is 0, the other assumes the carry-in is 1, with both additions performed in parallel. Once the correct carry-in is available from previous blocks, the correct sums are selected using multiplexors. Assuming the selection logic takes the same amount of time as a full adder, it follows that, for optimal performance, most of the blocks should be one bit longer than their immediate predecessors. An optimal design for a 16-bit adder would therefore be:

For an n-bit adder, the left hand block will need to be of size m, where $m + (m-1) + (m-2) + \ldots + 3 + 2 + 1 + 1 \geq n$. Noting that

$$\sum_{i=1}^{m} i = \frac{m(m+1)}{2}$$

we obtain

$$\frac{m(m+1)}{2} + 1 \geq n \quad \Leftrightarrow \quad m^2 + m - 2n + 2 \geq 0$$

Solving for the equality, we obtain

$$m = \frac{-1 \pm \sqrt{1 + 8(n-1)}}{2}$$

As $n \to \infty$, the positive solution approaches $\sqrt{2}\sqrt{n}$, so $m \geq \sqrt{2}\sqrt{n}$. Since the adder will take m time units to operate, its asymptotic time requirement is therefore $O(\sqrt{n})$. The asymptotic space requirement is clearly $O(n)$.

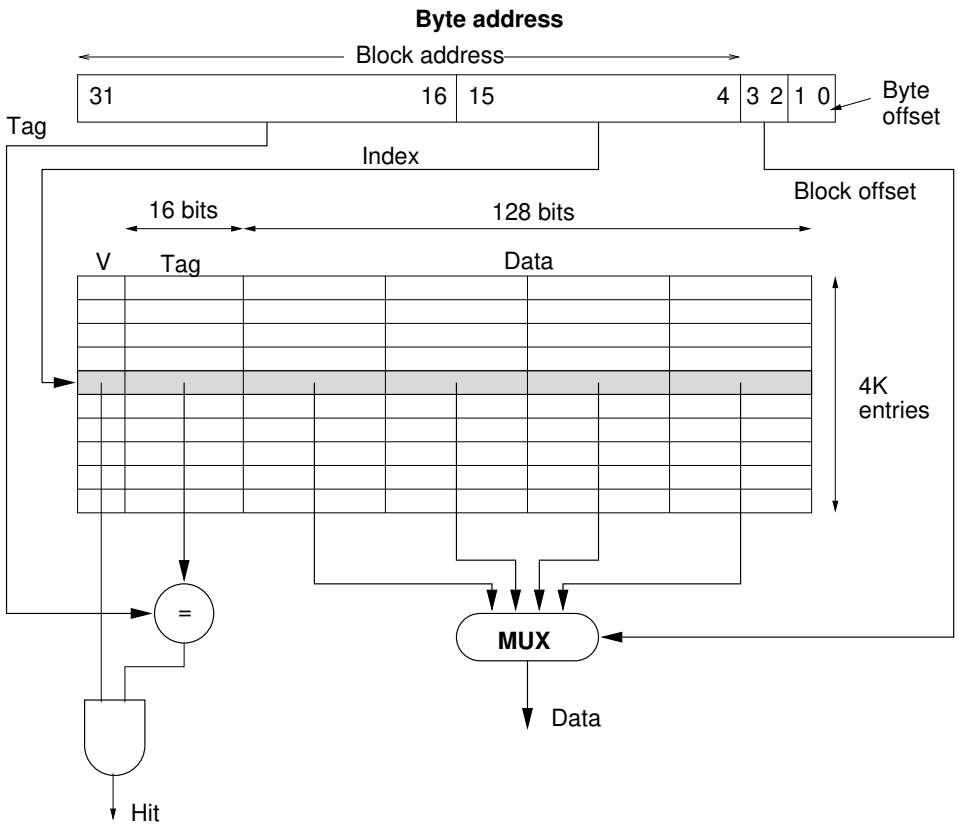
The asymptotic time and space requirements of a carry-lookahead adder are $O(\log n)$ and $O(n \log n)$ respectively. For large n, the carry-lookahead adder is therefore faster than both ripple-carry and carry-select adders, but it does require more space. [50%]

Assessor's remarks: An unpopular question that was nevertheless answered well by those candidates who attempted it. In (a), credit was awarded for sensible, intelligent discussions, of which there were many. All candidates identified the more limited number of registers available as MIP16e operands, but one or two failed to identify the obvious instruction fetching/caching advantages of MIPS16e. Responses to (b) were generally good, apart from a couple of candidates who confused carry-select adders with block carry-skip adders. The best answers provided a thorough justification of the $O(\sqrt{n})$ time complexity of carry-select adders.

2. **Caches and locality of reference**

(a) 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 block, thereby reducing the miss rate. [20%]

(b)



[30%]

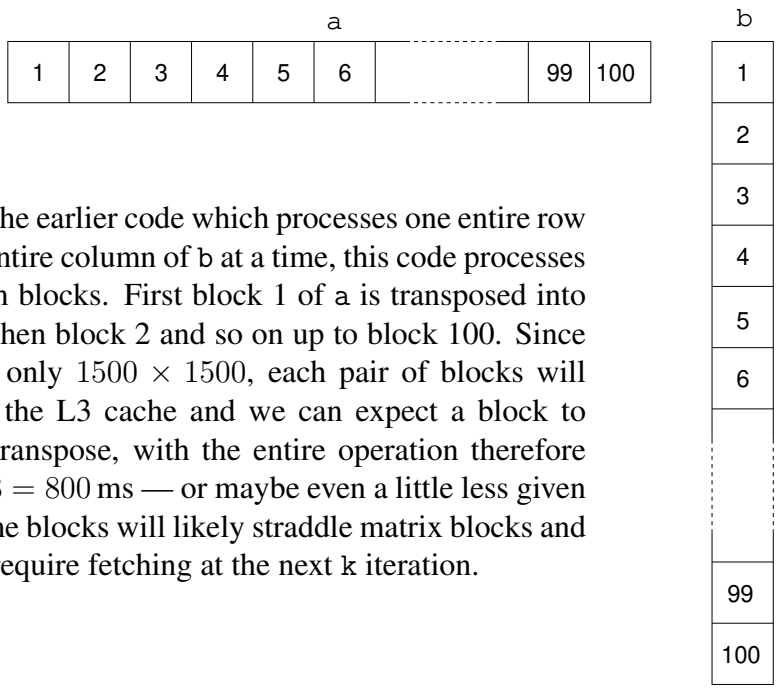
(c) Matrices are stored in memory row by row, so the elements of `a` are accessed sequentially inside the inner loop, with excellent spatial locality of reference and subsequent cache hits. The same cannot be said of `b`, where at each iteration of `i` there are `C` accesses at a stride of `R` bytes, before wrapping back to a similar set of addresses (just shifted along by one byte) at the next iteration of `i`.

When `R` = `C` = 1500, each matrix requires 2.25 MB of storage, so both will comfortably fit in the processor's L3 cache (the modest i5-7500T CPU used for the timings in the question has a L3 cache size of 6 MiB). So, assuming sufficient associativity, there should be no L3 cache conflicts and each block will be fetched from main memory just once. There should be no more than $2 \times 1500 \times 1500/k$ main memory accesses, where k is the block size.

However, when `C` = 150000, only about 1.4% of the two matrices will fit in the L3 cache, and there will therefore be a lot of block replacement. We do not wrap back to the beginning of `b` until 150000 accesses later, by which time the first block of `b` would most likely have been replaced in the cache. So we get mostly cache misses when accessing `b`, without the factor of k reduction in main memory accesses. This is why it takes around 250 times longer when `C` = 150000, even though there are only 100 times more operations. [25%]

(d) (i) The code segment calculates the transpose of the 1500×150000 matrix `a`, storing the result in the 150000×1500 matrix `b`. [10%]

(ii)



In contrast to the earlier code which processes one entire row of `a` and one entire column of `b` at a time, this code processes the matrices in blocks. First block 1 of `a` is transposed into block 1 of `b`, then block 2 and so on up to block 100. Since each block is only 1500×1500, each pair of blocks will fit entirely in the L3 cache and we can expect a block to take 8 ms to transpose, with the entire operation therefore taking $100 \times 8 = 800$ ms — or maybe even a little less given that some cache blocks will likely straddle matrix blocks and therefore not require fetching at the next `k` iteration.

[15%]

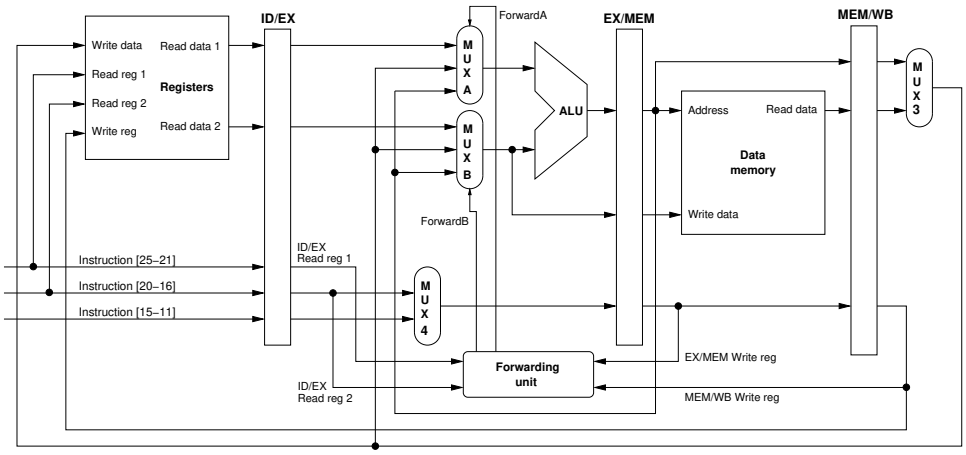
Assessor's remarks: This popular question tested the candidates' understanding of caches and how they might influence the way software is written. It was generally very well answered. In (a), almost all candidates offered perfect comparisons of direct-mapped and set-associative caches. In (b), the vast majority of candidates correctly identified the various bit fields of the address and how they were used in the cache. In (c), many candidates explained the timings convincingly, though some suggested that entire rows of the matrix could fit into a single cache block, while others neglected to reference the 800 ms expected run-time of the larger example given the number of elements to transpose. In (d), the vast majority of candidates identified the use of blocking to improve locality of reference.

3. **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)



The forwarding unit compares the registers that 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) The data in `x` is best thought of as a 512×512 matrix $\mathbf{X}$, with rows stored consecutively in the array. For $j \geq 1$, the code replaces $\mathbf{X}_{i,j}$ with $\mathbf{X}_{i,j-1}$, with the effect of copying the first element in each row of $\mathbf{X}$ to all the other elements in the row.

(i) The buggy compiler has produced code that increments the array index `$10` once for every element copied. This would have the effect of copying the top-left element of $\mathbf{X}$ into the following 512×511 subsequent elements, unlike the C++ code which does not copy the last element in a row into the first element of the next row. What is missing is an extra increment, i.e. `addi $10,$10,4`, immediately before the final `bne`, to move the index from the last element of a row to the first element of the next row.

While this was the only bug specifically asked about, there are others. `$8` and `$9` should be incremented/initialized by 1, not 4, since they represent `i` and `j` which count up to 512. [15%]

(ii) The pipeline will have to stall for three clock cycles after each branch instruction, since the PC is updated at the fourth pipe stage. For standard data forwarding to the ALU inputs as shown above, and assuming no delayed loads, the pipeline would also have to stall for one clock cycle between the `lw` and the `sw`, since the data is read from memory by the `lw` at the same time it is required at the ALU inputs by the `sw`. [15%]

(iii) Including the stalls, the inner loop as written would take 9 clock cycles to execute. We could avoid the stall between the `lw` and `sw`, and hence reduce the inner loop execution time to 8, by moving one or both of the `addi` instructions to before the `sw`. Unrolling the inner loop by a factor of n would reduce the number of `addi` and branch instructions (and the three subsequent stalls) by the same factor. Assuming the less compact code still fits in the instruction cache, this would improve the performance significantly (e.g. for $n = 2$, one iteration of the unrolled loop would take 10 clock cycles compared with $2 \times 8 = 16$

previously. Switching the C++ loops would produce the same result but would destroy spatial locality of reference to `x`. Unless the compiler were smart enough to spot this and switch them back (highly unlikely), we would expect significantly impaired performance, with the compiled MIPS instructions striding through the data 512 words at a time instead of one word at a time, resulting in a high L1/L2 cache miss rate. [30%]

Assessor's remarks: This question tested the candidates' understanding of pipelining, hazards, compiler optimizations and locality of reference. It was generally very well answered. In (a), almost every candidate could explain perfectly the principles and complications of pipelining. In (b), most candidates offered a good schematic of the data forwarding unit, though some did not highlight the essential three-input multiplexors at the ALU inputs. In (c)(i), around a third of the candidates correctly identified the missing increment of `$10`. The rest of part (c) was not dependent on (i), with many candidates offering excellent analysis of the stalls and possible optimizations, though there was some inappropriate discussion of delayed loads given that the question stated that branch hazards are resolved by stalling. It was good to see almost all candidates correctly context switch from pipeline hazards to spatial locality of reference, when considering switching the order of the C++ loops, The very best responses identified the possibility of instruction cache misses with too much loop unrolling.

Andrew Gee
May 2024