(a) [Computer Arithmetic] (0.75pts) The chief architect of the MN-4363 processor, Tim FlimFlam, is exploring the ISA design space.

Tim claims that there is no need for a multiply instruction. A left shift should serve the purpose. One of the team members, Billy ShillyShally warns Tim that it is not left shift, but left shift along with add that would suffice to implement multiply.

- Would Tim's approach work? Explain.

  By relying on a left-shift, Tim can only multiply by 2^i, $i = 0, 1, 2, 3, ...$, hence only a subset of multiplications can be taken care of with his approach.

- Would Billy's approach work? Explain.

  Provided that (i) each number can be represented as $\sum 2^i + r$, with $\sum 2^i$ assuming the minimum possible number of terms, and r corresponding to a reminder which would be 0 for even, 1, for odd numbers; (ii) multiplication is distributive over addition, Billy's approach would work.

  "$\sum 2^i$ assuming the minimum possible number of terms" translates into $i = 1, 2, 3, ...$ taking its maximum possible value(s).

  In this case, to multiply two numbers X, and Y, first one should be picked (say X) for decomposition, i.e. to be represented as $\sum 2^i + r$. Then, all we need to do is, for each term in $\sum 2^i$, shifting Y by i, and adding all shifted Y and r together to arrive at the final product.

  An analogous discussion applies if Y is selected for decomposition. While the number picked for decomposition is arbitrary for *correctness* of multiplication, this selection may affect the number of necessary additions and shifts. Hence, depending on the cost of addition vs. shift, this selection may have an impact on performance/power consumption.

- Demonstrate how two 16-bit numbers, $(129)_{10}$ and $(35)_{10}$ can be multiplied using Billy's technique.

  If we pick $(129)_{10}$ for decomposition, $(129)_{10} = 128 + 1 = 2^7 + 1$ implies $i = 7$ and $r = 1$. Hence, to arrive at the product, we need to shift $(35)_{10}$ by $i = 7$ and and add $r = 1$ to it.

  An analogous discussion applies if $(35)_{10}$ was selected for decomposition.

(b) [ISA] (1.5pts) The chief architect of the MN-4363 processor, Tim FlimFlam, is exploring the ISA design space.

(i) Tim plans to restrict the instruction word to 32 bits, but wants to design a jump instruction that can change all bits of the PC. Their ISA already assumes byte-addressing and 32-bit wide general purpose registers (and PC). This is the most effective way to handle long jumps, Tim claims.

- Can Tim's approach work? Explain.

A single instruction to change all bits of the PC would be effective in implementing long jumps. If the instruction word is restricted by N bits, and so is PC, this single instruction cannot assume any variant of direct addressing (compare to pseudo-direct addressing in MIPS, e.g.) because some bits of the instruction word should be allocated for the op-code (jump) itself. A possible solution in this case appears to be a "jump (jump-)register" instruction (similar to MIPS *jr*), which, when encountered, would overwrite the contents of the PC with the value of the jump-register. This jump-register can point to any general purpose register, hence would contain an N-bit value.

(ii) Tim's team member Billy ShillyShally, on the other hand, claims that a long jump can be implemented as a chain of MIPS-like pseudo-direct jumps and conditional branches.

- Can Billy's approach work? Are conditional branches required? Explain.
- How does Billy's "solution" compare to Tim's?

Under MIPS-type pseudo-direct addressing, the 26-bit immediate field of the (32-bit) jump instruction is interpreted as a word offset, hence multiplied by 4 (to arrive at an 28-bit constant), and concatenated with the higher order 4 bits of the current PC (to render an 32-bit address).

For conditional branches, the immediate field has only 16 bits, gets interpreted as a word offset (similar to jumps), and added to the current PC.

- A "long" jump can be implemented by a chain of pseudo-direct jumps and conditional branches: Each pseudo-direct jump will be limited by the higher order 4 bits of the PC which remain intact across jumps. Each jump would cover a range of 2^{28}B(ytes) = 256MB. This means that *a chain of jumps only can never get out of a program region bounded by the higher order 4 bits of the PC.* Conditional branches, on the other hand, can possibly break such boundaries, although they do not necessarily provide a wide branching range relative to PC (by 2^{16} instructions roughly). This is because the addition of the 18-bit constant to PC can change the most significant bits of PC.

  If a jump target resides further than the maximum possible reach by pseudo-direct addressing, say R, a series of jump/branch pairs can extend the range, provided that each branch following the jump can reach passed a 256MB program region. At the target of the branch, the next jump/branch pair would reside. All we need is decomposition of $R = \sum R_i$, $i = 1, 2, 3, ...$, with each R_i remaining within reach per jump or conditional branch. To reduce the number of hops, we can pick the maximum possible R_i per hop.

  Since jumps do not depend on a condition, conditional branches included in the chain should be adjusted to have the condition evaluate to true. The composition of R affects the number of hops, hence the instruction count, which in turn determines performance.

- Tim's *jr* variant provides a possibly larger jump range R at the cost of a single instruction only – wouldn't incur any decomposition (i.e. how R should be synthesized from per jump/branch ranges) overhead or additional instructions.

(c) [Branch Prediction] (1pt) The chief architect of the MN-4363 processor, Tim FlimFlam, is now working on the branch predictor. The processor relies on two separate structures: A prediction table to keep track of branch direction, and a BTB (Branch Target Buffer) to keep track of branch target addresses. Tim claims that, in processing a branch, if there is a miss in the BTB but a corresponding entry in the prediction table does exist, the instruction fetch should continue from PC+4, i.e. the fall-through path of the branch, since the predictor was likely wrong but BTB correct.

- Under what circumstances can this be the case? Provide at least two possible scenarios.
- Do you agree with Tim's solution? Explain.

Hint: BTB can be thought of as a cache, with limited number of entries. Branches constitute usually less than 20% of the instruction mix. A BTB miss means either (i) the instruction is not a branch (in which case the branch predictor never contained any info about the instruction) or (ii) the instruction is a branch, but is a cold miss (in which case the branch predictor is also going to be cold with respect to this branch) or (iii) the instruction is a branch that got kicked out of the BTB for capacity/conflict reasons, the predictor has many more entries than the BTB, so the predictor is somewhat more likely to be right and the BTB wrong. Nonetheless, case (i) is much more likely than (iii) because branches are usually less than 20% of the instruction mix.

(d) [Quantitative Analysis] (0.75pts) As the chief architect of Massively Parallel Inc., you hired a summer intern called John Gustafson. He was in charge of characterizing the speedup of one of your Massively Parallel applications as a function of the number of processors, N, devoted to computation.
For N=1024, he claimed that he was able to observe a speedup of $\approx 512\times$ over the sequential version of the code (N=1). This result remains much larger than the speedup claimed by your previous intern, Gene Amdahl.
Assumption: The overall execution time of this code for N=1 would be s+p, where p corresponds to the section of code that can be parallelized. On N processors, this section would take $p/N <$ time units $>$. $s + p = 1 <$ time unit $>$. s does not change with N.

(i) Gene's result was in-line with the speedup Amdahl's Law suggests. Derive the speedup as a function of p for N=1024. What should p be to achieve a speedup $\approx 512\times$?

$Speedup = (s+p)/(s+p/N) = 1/(1-p+p/N) = 1/(1+p(1/N-1))$. $p \approx 0.999$ $for\ Speedup = 512\times$.

(ii) You find out that John approached the task in a different way: For Gene, the goal was reduction of (parallel) execution time for the *same* quantity of (total parallel) work (i.e. constant p). John, on the other hand, interpreted *speedup* as the increase in the (total parallel) quantity of work for the same (parallel) execution time. Further, he assumed that the quantity of work increases linearly with the number of processors N (and that the execution time is proportional to the quantity of work). If, for any N$>$ 1, $s_{Gustafson} + p_{Gustafson} = 1$, derive the speedup as a function of $p_{Gustafson}$ and N – *Gustafson's Law*[1]. What should $p_{Gustafson}$ be to achieve a speedup $\approx 512\times$?

$Speedup = (s_{Gustafson} + N \times p_{Gustafson})/(s_{Gustafson} + p_{Gustafson}) = (1 - p_{Gustafson} + N \times p_{Gustafson})/1 = 1 + p_{Gustafson}(N-1)$. $p_{Gustafson} \approx 0.4995$ $for\ Speedup = 512\times$.

(iii) You are now preparing a manual for your next intern. When should the new intern rely on Gustafson's Law, when on Amdahl's? The answer is already provided in (ii). Depends on how speedup is defined. If the work (problem size) is constant, and the execution time per processor reduces as more processors becomes available, Amdahl's Law applies. Otherwise, if the work (problem size) increases with increasing number of processors, Gustafson's Law applies. In this case, the execution time per processor remains constant.

[1]"Gustafson Law" or "Gustafson-Barsis Law" indeed exists.