Subleq$_{\ominus}$: An Area-Efficient Two-Instruction-Set Computer

Noriaki Sakamoto[†] Tanvir Ahmed[†] Jason Anderson[‡] Yuko Hara-Azumi[†*]
[†]Tokyo Institute of Technology [‡]University of Toronto [*]JST PRESTO

***Abstract*—Applications with strict resource/power constraints demand the research and development of area-efficient processor designs that deliver reasonably good performance with small circuit area. While the ARM and RISC-V [1] ISAs are lightweight alternatives to x86, they nevertheless consume considerable circuit area and power. In this paper, we return to a fundamental question: how area efficient can a processor be while retaining the property of being "Turing Complete" (i.e., capable of realizing *any* computation)? Beginning with a recently published one-instruction-set computer (OISC), that uses a minimal amount of resources, we consider adding a second instruction to the instruction set and justify the choice of such an instruction. An experimental study illustrates the benefits of our ISA extension in terms of performance at minimal area cost.**

I. Introduction

With increased diversity and complexity of applications, processors have been augmented with richer instruction sets for better computational efficiency. This is true not solely for general-purpose processors, but also for embedded microprocessors. For example, even one of simplest microprocessors, the Cortex-M0, employs the ARMv6-M instruction set architecture (ISA), which supports 56 instructions for general data processing and I/O control [2]. Historically, Dennard scaling has reduced the necessity for very area-efficient ISAs, as migration to the next process node assured better power efficiency per transistor. However, such scaling has stalled, and with recent trends towards ultra-low-power Internet-of-Things embedded applications, there is a need to re-visit the utility of low-area-cost processor ISAs. Towards this objective, we explore processors with very few instructions and low area.

Current RISC processors, with dozens of different instructions, are a distant cousin of One Instruction-Set Computer (OISC) architectures [3], [4], which have a single instruction capable of realizing *any* computation. OISC architectures sacrifice operational efficiency for very low area. This motivated us to consider the question: what ISA extensions can be made to an OISC ISA to raise performance with minimal damage to area? As a first step, this article proposes a Two-Instruction-Set Computer (TISC), based on an OISC, where a second instruction is judiciously added to improve efficiency, while retaining a low area footprint. While traditional ISA extensions directly speed up specific computational hotspots (e.g., patterns of frequently executed instructions) by introducing new resources [5], our work resolves the underlying cause of slowness in a recently published OISC architecture. Thus, our approach effectively speeds up different types of hotspots by a single new instruction. Moreover, to the extent possible, the new instruction reuses the resources of the base processor to mitigate area overhead.

In this paper, we target the `subleq` OISC ISA, whose instruction performs subtraction followed by a conditional jump, as the base processor. We add a bit-reversal `subleq` instruction in order to speed up a variety of arithmetic operations. The extended `subleq`, named Subleq$_{\ominus}$, flexibly selects the faster instruction out of the two (the original `subleq` or the bit-reversal `subleq`). In an experimental study, we demonstrate the effectiveness of the proposed TISC ISA in achieving good speedup (2.78× on average, and up to 7.75×, which outperforms traditional approaches of ISA extension achieving only up to 3.56×) and mitigation of area overhead (1.33× higher area, whereas traditional approaches introduced 1.87× and 5.46×) at the same time.

Aside from OISC ISA extension, our work is applicable to other types of processors, including those used in current commercial products. Thus, we expect this work is of keen interest to both academic, as well as industrial computer architects.

II. Subleq Computer

The rationale for RISC ISAs is to keep processors small by limiting the number of instructions. The ultimate RISC ISA is a single instruction [3], [4]—an OISC ISA—where all computational work is expressed as a sequence of the single instruction. The processor's structure is kept simple and small; however, the computational work is generally inefficient as a consequence of the number of instructions that are needed to get work done. Subleq is an OISC employing word-addressable memory. The instruction is `subleq`, whose mnemonic is "`subleq` A, B, C"—subtraction of the first two operands, comparison of the subtraction's result with zero, and then jump to an address (the subsequent instruction in memory or the third operand) according to the comparison result [3]. The semantics are as follows:

$$\begin{aligned} r &\leftarrow \mathrm{mem}[B] - \mathrm{mem}[A] \\ \mathrm{mem}[B] &\leftarrow r \\ PC &\leftarrow \begin{cases} C & \text{if } r \leq 0 \\ PC + 3 & \text{otherwise} \end{cases} \\ &\textbf{halt} \text{ if } C < 0. \end{aligned}$$

The condition $r \leq 0$ is equivalent to "$r = 0$ or r's MSB is equal to 1." Different types of operations (add, subtract, multiply, logical, etc.) will require different numbers of `subleq` instructions to execute. For example, an addition operation takes five `subleq` instructions. The specific recipe of `subleq`

```
1 for (int i = 0; i < w - 1; i++)
2   x <<= 1;
3 if (x < 0)
4   lsb = 1;
5 else
6   lsb = 0;
```

Fig. 1: LSB search for input x (w is word size).

instructions to perform addition, test on the MSB, left shift, and the other operations is omitted for space considerations and the interested reader is referred to [3] and [6].

As is apparent above, each instruction requires subtraction, conditional jump on zero/sign, and self modification. Thus, for performing multiplication, shifts, and bitwise logical operations, a loop is executed w (word-length) times in order to access bitwise information from MSB to LSB, via repeated left shifting, as shown in Fig. 1. Essentially, the LSB must be shifted into the MSB position, making it the sign bit, thereby allowing the Subleq's comparison-with-0 functionality to be leveraged. Left shift can be implemented with Subleq by adding an operand to itself, which is possible by first creating a negated version of the operand, and then using the existing subtraction capability, i.e., $Z \ll 1 \equiv Z - (-Z)$.

The second column of Table I shows the time complexity of commonly occurring operations when they are executed with `subleq` instructions. One can observe that shift, multiplication and logical operations are quite inefficient as compared with addition, as their time complexity depends on the shift amount (n) or word-width (w).

III. TISC: Two-Instruction-Set Computer

A. Subleq$_\ominus$: Extended Subleq

To reduce the inefficiency of the Subleq OISC ISA for operations like multiplication and shift, we propose to add a bit-reversal `subleq` instruction, whose purpose is to speedup the LSB search. We refer to this as extended Subleq, Subleq$_\ominus$. The instruction of Subleq$_\ominus$, `subleq`$_\ominus$, consists of two operations: 1) subtraction and branch on sign, and 2) bit-reversal subtraction and branch on evenness, as follows:

$$r \leftarrow \begin{cases} \mathrm{mem}[B] - \mathrm{mem}[A] & \text{if } C > 0 \\ \ominus(\ominus\mathrm{mem}[B] - \ominus\mathrm{mem}[A]) & \text{if } C < 0 \end{cases}$$

$$\mathrm{mem}[B] \leftarrow r$$

$$PC \leftarrow \begin{cases} |C| & \text{if } C > 0 \text{ and } r \leq 0, \\ & \text{or } C < 0 \text{ and } \ominus r \leq 0 \\ PC + 3 & \text{otherwise} \end{cases}$$

halt if $C = 0$.

where $\ominus$ is a monadic operator (bit reversal), e.g., $\ominus$`3'b001 = 3'b100` and $\ominus$`4'b1010 = 4'b0101`. Observe that, as mentioned above, the second branching condition reduces to checking evenness because bit reversal swings the MSB into the LSB position (thereby indicating even/odd). Note that the more efficient instruction is selected, `subleq` or `subleq`$_\ominus$, depending on the operation and the value.

Comparing the original Subleq and the proposed Subleq$_\ominus$, the two variants both require the hardware capability for subtraction and comparison. The additional hardware required in Subleq$_\ominus$ is primarily multiplexers that permit the selection between either normal (i.e., unreversed) or bit-reversed operands. In hardware, bit-reversing of operands is simply permuted wiring. While the original `subleq` instruction is efficient at 1-bit left shift, the bit reversal means the new `subleq`$_\ominus$ instruction is efficient at 1-bit right shift.

B. Reduction of Time Complexity

With Subleq$_\ominus$, the time complexity for performing arithmetic operations is significantly reduced. We first illustrate this using the computation of logical right shift (`srl rd,rt,n` performing `rd = rt` $\gg$ `n`; same as MIPS's Shift Left Logical instruction) in Subleq compared with Subleq$_\ominus$. For this instruction, Fig. 2 and Fig. 3 give C-like pseudo code using 1-bit shift left and right, respectively (in the following pseudo codes, we use `rs` and `rt` as source registers and `rd` as a destination register, similarly as MIPS). Both of them get input values from memory location `rt` and `n` and write back output into memory location `rd`. When 1-bit left shift must be used (Fig. 2), the approach is to extract the $(w-n)$ higher-order bit positions of `rt` (by iteratively extracting the MSB leveraging the available left shift) and move these bits into the lower-order bit positions of `rd`. Figure 4 shows a brief sketch of how the algorithm works: by repeating 1-bit logical left shift on a double word $\{$`rd,rt`$\}$, we can implement $(w-n)$-bit logical left shift (in the bottom of Fig. 4), whose upper result equals to the result of n-bit logical right shift (in the top of Fig. 4). When 1-bit right shift is realized by Subleq$_\ominus$, its time complexity reduces from $O(w)$ to $O(1)$. For an n-bit right shift (n is the shift amount), Subleq$_\ominus$ can select faster among 1-bit left shift and right depending on the shift amount (n) with additional instructions which can be implemented with no additional hardware, comparing therefore reducing the time complexity from $O(w)$ to $O(\min(n, w-n))$.

Similarly, multiplication is defined as: `mult rt,rs`, which performs $\{$`hi,lo`$\}$ = `rt` $\times$ `rs`, where "$\{$`hi,lo`$\}$" refers to a double word whose upper and lower words are `hi` and `lo`. It can be calculated in different ways, depending if only efficient left shift is available (Fig. 5) or if *both* efficient left shift and right shift are available (Fig. 6). Note that Subleq$_\ominus$ always selects that latter version that uses 1-bit right shift, which is faster, reducing time complexity from $O(w)$ to $O(l)$ (l is bit-width of the operand, i.e., $l = \lceil \log_2 \mathtt{rs} \rceil$). With reference to Fig. 5, checking the MSB of multiplier `rs`, the algorithm shifts `rt`, add the multiplicand to the sum $\{$`hi,lo`$\}$, and shifts the sum. The loop is executed w times.

In the algorithm that requires right shift, shown Fig. 6, multiplicand (`rt` = $\{$`mh,ml`$\}$) is shifted. Checking the LSB of multiplier `rs`, the algorithm shifts the multiplicand (`rt` = $\{$`mh,ml`$\}$), and adds the shifted multiplicand to the sum $\{$`hi,lo`$\}$. The algorithm is similar to how long multiplication is taught in grade school. It is possible to exit the loop when the multiplier becomes zero to skip unneeded calculation. The same approach can be applied to logical operations, whose explanation is omitted for space considerations.

The time complexity of Subleq$_\ominus$ is summarized in the third column of Table I. Also, Figs. 7 and 8 compare the number of instructions for computing logical right shift and multiplication, respectively, by Subleq and Subleq$_\ominus$. The X-axis is the value of shift amount (n) in Fig. 7 and the bit-width

```
1  rd = 0;
2  for (int i = 0; i < w - n; i++) {
3      // shift double word {rd,rt} left by 1 bit
4      rd <<= 1;
5      rd += msb(rt);
6      rt <<= 1;
7  }
```

Fig. 2: Logical right shift by 1-bit left shift (available for Subleq and Subleq$_\ominus$).

```
1  rd = rt;
2  while (--n >= 0) // loop for n times
3      rd >>= 1;      // O(1) with subleq⊖
```

Fig. 3: Logical right shift by 1-bit right shift (available for Subleq$_\ominus$ only).

TABLE I: Comparison of time complexity (w: word-width, l: bit-width of the operand, and n: shift amount)

	Subleq	Subleq$_\ominus$
Addition	$O(1)$	$O(1)$
1-bit left shift	$O(1)$	$O(1)$
n-bit left shift	$O(n)$	$O(\min(n, w-n))$
1-bit right shift	$O(w)$	$O(1)$
n-bit right shift	$O(w)$	$O(\min(n, w-n))$
Multiplication	$O(w)$	$O(l)$
Logical	$O(w)$	$O(l)$

of multiplicator (l) in Fig. 8, and the y-axis is the number of `subleq` or `subleq`$_\ominus$ instructions. Dots in the figures reflect 10,000 randomly-generated values. Solid and dotted lines represent the average for each x-value. For logical right shift, use of 1-bit right shift (left) is faster when n is small (large), and for multiplication, use of 1-bit right shift is always faster. When n is large (especially $n = 17$ to 20), Subleq$_\ominus$ takes slightly more instructions due to the overhead of dynamic selection of the faster algorithm. However, as will be shown in the next section, this overhead is negligible compared with the speedup achieved for the other computations during the execution of actual applications.

IV. Experimental Study

We implemented OISC and TISC architectures in `C` and synthesized RTL descriptions using commercial tools (Vivado HLS 2015.1 and ISE 13.4) targeting the Xilinx Virtex 6 FPGA (`xc6vsx475tff1759-2`) to evaluate the circuit area and clock frequency. Also, we evaluated performance (i.e., dynamic instruction counts) for a number of practical applications using a simulator developed in house. The benchmarks were compiled by GCC-4.1.1 with -O2 optimization. For more detail, please refer to [6] for a complete description of the compilation flow.

Four architectures were considered: Subleq (baseline), two Subleq-based TISCs, and Subleq$_\ominus$. All of them have CPI of 4. The two TISCs are representative of a traditional approach to ISA extension, wherein an instruction is added to speed up specific hotspot operations. In particularly, *TISC*$_\mathrm{s}$ and *TISC*$_\mathrm{m}$ incorporate two-operand right shift and multiplication, respectively, as the second instruction, since they are both hotspot operations (and very time intensive when only `subleq` is available), Meaning, the *TISC*$_\mathrm{s}$ and *TISC*$_\mathrm{m}$ architectures contain dedicated hardware for right shift and multiplication, respectively.

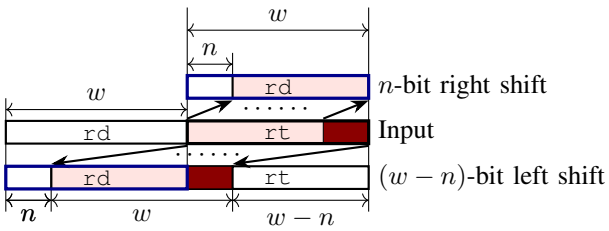

Fig. 4: n-bit right shift (in Fig. 3) and $(w-n)$-bit left shift (in Fig. 2).

```
1  hi = lo = 0;
2  for (int i = 0; i < w; i++) { // loop for w times
3      {hi, lo} <<= 1;
4      if (msb(rs) > 0)
5          {hi, lo} += rt
6      rs <<= 1;
7  }
```

Fig. 5: Multiplication by 1-bit left shift (available for Subleq and Subleq$_\ominus$; however, Subleq$_\ominus$ always selects 1-bit right shift)

```
1  hi = lo = 0; {mh, ml} = {0, rt};
2  // {mh,ml} is a double-word multiplicand and repeatedly shifted
3  while (rs != 0) { // loop for l(= ⌈log2 rs⌉) times
4      if (lsb(rs) == 1)     // O(1) with subleq⊖
5          {hi, lo} += {mh, ml};
6      {mh, ml} <<= 1;
7      rs >>= 1;             // O(1) with subleq⊖
8  }
```

Fig. 6: Multiplication by 1-bit right shift (available for Subleq$_\ominus$ only)

TABLE II: Synthesis Results

Architecture	Subleq	*TISC*$_\mathrm{s}$	*TISC*$_\mathrm{m}$	**Subleq**$_\ominus$
#LUTs	147	275 (×1.87)	862* (×5.86)	195 (×1.33)
$f_{\max}$ [MHz]	166.9	132.7 (×0.80)	84.0 (×0.50)	158.1 (×0.95)

* 180 LUTs, 3 DSP48s

A. Experimental Results

Table II shows the synthesis results (i.e., circuit area [LUTs] and clock frequency). Numbers in parentheses represent ratios relative to the baseline (Subleq). Note that *TISC*$_\mathrm{m}$ uses not only LUTs but also three DSP units, however, the results in the table show the equivalent number of LUTs permitting comparison with the other architectures. As shown in the table, Subleq$_\ominus$ has less overhead in both circuit area and frequency than the two TISCs with hardened instructions. This is because Subleq$_\ominus$ can reuse most of the original Subleq, while the two TISCs require appreciable new resources for the second instruction (right shift or multiplication). Note that *TISC*$_\mathrm{m}$ has bigger overhead in clock frequency because of multiplication. The Subleq$_\ominus$ requires ∼50 more LUTs than Subleq and operates at roughly the same clock frequency. We believe this is a modest area cost, considering the performance benefits it affords.

Table III compares the wall-clock time (i.e., [dynamic cycle counts] × [clock period]) of 12 applications executed by the four architectures. The first column lists the benchmarks used, and the second to fifth columns show the wall-clock time of the four architectures for each benchmark. Geomean is shown in the bottom row. Numbers

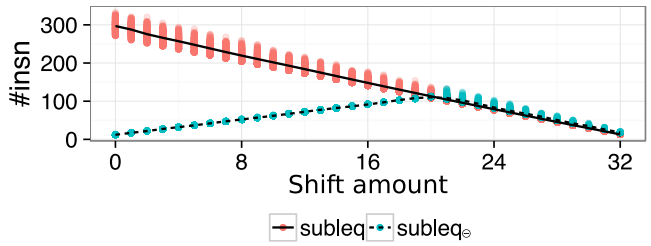

Fig. 7: Number of instructions for logical right shift

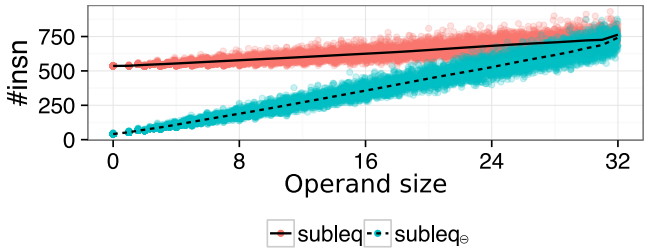

Fig. 8: Number of instructions for multiplication

in parentheses are speedup ratios vs. Subleq. Note that all of these benchmarks require right shift, while only `adpcm`, `bubble`, `gsm`, `intmm` and `jfdtint` have multiplication. Although *TISC*$_{\mathrm{s}}$ also reduced dynamic cycle counts of its target hotspot operations, as seen from the table, its wall-clock time degrades relative to the original Subleq. This is because the overhead in clock period is larger than reduction in dynamic cycle counts. Whereas *TISC*$_{\mathrm{m}}$ achieved a speedup for multiplication-contained applications, it rather provided speed degradation for multiplication-less ones due to clock period elongation without any cycle-count benefit. Conversely, Subleq$_{\ominus}$ achieves significant speedup in all benchmarks—on average 2.78× and up to 7.75×. Interestingly, Subleq$_{\ominus}$ achieved the largest speedup (7.75×) in `intmm`, which has the highest ratio of multiplication, and outperformed even *TISC*$_{\mathrm{m}}$ (only 3.83×). Recalling the fact that the clock degradation is small in Subleq$_{\ominus}$, these speedup effects come from reduction in cycle counts. In order to confirm this, Fig. 9 shows cycle counts for four benchmarks, selected to reflect a diversity of operations types: `bf`, `gsm`, `jfdctint`, and `mpeg`. While the two TISCs can reduce dynamic cycle counts in only limited types of operations, Subleq$_{\ominus}$ provides cycle-count reductions across varied operation types, thereby benefiting a wider scope of applications.

Our evaluation demonstrated that Subleq$_{\ominus}$ achieves considerable speedup for a variety of applications irrespective of the dominant type of instructions. The implementation results show that Subleq$_{\ominus}$ realizes an ISA extension which is still simple yet area-efficient. The results highlight the value of improving instruction inefficiency, while re-using a significant portion of the baseline processor.

V. Conclusions

In this paper, we propose to extend the recently published OISC ISA, Subleq, with an additional instruction, which re-uses the already existing hardware, with the operands optionally bit-reversed. The extension targets the weakness of Subleq in handling right-shift, multiplication, and logical operations, which require many instructions to realize. An experimental study demonstrated the area bloat and clock period increase associated with the extension to be quite small. Across a set of benchmarks, the proposed TISC offers a 2.78× wall-clock-time speedup, on average, relative to the original Subleq. As future work, we intend to explore area-efficient ISA extensions by adopting additional instructions. We would also like to explore the power and energy consumed by the various processor architectures.

TABLE III: Wall-Clock Time and Speedup Ratio

Benchmarks	Subleq	*TISC*$_{\mathrm{s}}$	*TISC*$_{\mathrm{m}}$	**Subleq**$_{\ominus}$
adpcm	27.68 ms	32.25 ms (0.86)	7.59 ms (3.64)	5.28 ms (5.24)
bf	309.55 ms	325.44 ms (0.95)	382.20 ms (0.81)	171.03 ms (1.81)
bs	7.61 µs	6.43 µs (1.18)	4.10 µs (1.86)	2.11 µs (3.61)
bubble	230.39 ms	289.53 ms (0.80)	106.45 ms (2.16)	54.46 ms (4.23)
crc	9.86 ms	10.60 ms (0.93)	15.51 ms (0.64)	6.32 ms (1.56)
fibcall	3.43 µs	4.31 µs (0.80)	4.69 µs (0.73)	2.60 µs (1.32)
gsm	11.14 ms	11.36 ms (0.98)	9.68 ms (1.15)	5.51 ms (2.02)
insertsort	136.64 µs	171.82 µs (0.80)	52.67 µs (2.02)	26.21 µs (5.21)
intmm	187.14 ms	233.56 ms (0.80)	48.81 ms (3.83)	24.16 ms (7.75)
jfdctint	738.36 µs	825.49 µs (0.89)	597.10 µs (1.24)	293.67 µs (2.51)
mpeg	4.69 ms	3.87 ms (1.21)	5.24 ms (0.89)	2.44 ms (1.92)
vecadd	2.76 ms	3.47 ms (0.80)	3.16 ms (0.87)	1.50 ms (1.84)
geomean	3.25 ms	3.59 ms (0.91)	2.30 ms (1.41)	1.17 ms (2.78)

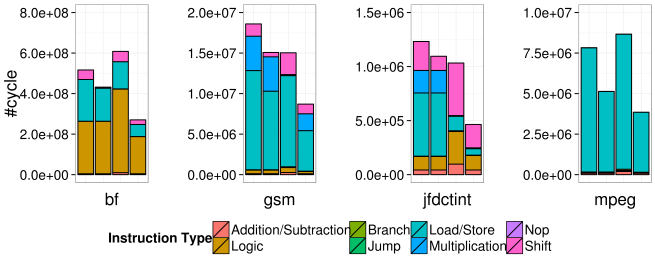

Fig. 9: Dynamic instruction count for four benchmarks; bars from left-to-right: Subleq, *TISC*$_{\mathrm{s}}$, *TISC*$_{\mathrm{m}}$, and Subleq$_{\ominus}$.

References

[1] *RISC-V Foundation*, http://riscv.org/, 2016.

[2] *ARMv6-M Architecture Reference Manual*, ARM, 2010.

[3] O. Mazonka, "Bit copying: The ultimate computational simplicity," *Complex Systems*, vol. 19, no. 3, 2009.

[4] D. W. Jones, "The ultimate RISC," *SIGARCH Comput. Archit. News*, vol. 16, no. 3, pp. 48–55, Jun. 1988.

[5] C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," *ACM Trans. Reconfig. Technol. Syst.*, vol. 4, no. 2, pp. 1–28, 2011.

[6] T. Ahmed *et al.*, "Synthesizable-from-C embedded processor based on MIPS-ISA and OISC," in *International Conference on Embedded and Ubiquitous Computing (EUC)*, Oct. 2015, pp. 114–123.