Synthesizable-from-C Embedded Processor Based on MIPS-ISA and OISC

Tanvir Ahmed[†], Noriaki Sakamoto[†], Jason Anderson[‡], and Yuko Hara-Azumi[†,*]
[†]Dept. of Communications and Computer Engineering, Tokyo Institute of Technology
[‡]Dept. of Electrical and Computer Engineering, University of Toronto
[*]JST, PRESTO
[†]{tanvira|noriakis|hara}@cad.ce.titech.ac.jp, [‡]janders@ece.utoronto.ca

***Abstract*—We describe a lightweight open-source MIPS-ISA processor, wherein performance and area can be flexibly traded-off with one another. The processor contains an ultra-low-cost co-processor capable of executing programs comprised of `SUBLEQ` instructions (subtract and branch if the difference is $\leq$ 0), which recent work has shown to be sufficient for *any* computation. Area/performance trade-offs are realized by implementing a user-selectable subset of MIPS instructions with functionally equivalent `SUBLEQ` sub-routines that run on the co-processor. Silicon area is reduced as more MIPS instructions are implemented with the co-processor, rather than "natively" using functional units within the host MIPS. The processor is described in the `C` language and synthesized to an FPGA hardware implementation with high-level synthesis (HLS). Since it is specified at a high level of abstraction, it is straightforward to tailor to any application. As such, the processor can be viewed as a family of processors with different area/performance/power characteristics. In an experimental study, we compare a variety of processor variants, wherein different subsets of MIPS instructions are handled by the co-processor. We also compare the proposed synthesizable processor with a hand-designed 5-pipeline-stage MIPS implementation, and achieve area reductions ranging from 2.5–4×.**

***Keywords*—Coprocessors, Field programmable gate arrays, Embedded Processor**

I. Introduction

Recent trends in computing have seen explosive growth in the low-cost/low-power embedded context, where lightweight processors are ubiquitous in varied applications, such as wearable, sensor, automotive, and home appliance. In this domain, processor cost and power are often the key drivers. However, the amount of performance one is willing to sacrifice for a reduction in cost/power is highly application dependent. What is needed is a flexible processor, wherein cost (silicon area) can be traded-off with performance in a straightforward way, while at the same time supporting a standard instruction set architecture (ISA) targetable with mainstream compiler technology. In this paper, we propose an open-source MIPS-ISA processor with a unique architecture that permits such performance/cost trade-offs. Our processor is described in the `C` language and synthesizable with a state-of-the-art high-level synthesis (HLS) tool [1].

At the extreme end of the low-cost computing spectrum is the *one instruction-set computer* (OISC), comprising an ISA with a single instruction, thereby eliminating the need for an opcode. A recent example of this is the `SUBLEQ`-based OISC machine [2]: it supports a single ternary instruction with arguments, a, b, and c. The instruction behavior is as follows: the value at address a is subtracted from the value at address b and the result is stored in address b. If the difference is $\leq$ 0, control transfers to address c. An OISC with `SUBLEQ` is proven to be Turing complete — it is capable of performing *any* computation.

We propose a novel processor architecture that supports the MIPS ISA and contains within an OISC `SUBLEQ` co-processor. The implementation of each MIPS arithmetic/logical instruction can be designated as: 1) native, or 2) low-cost. Instructions designated as native are implemented in the traditional way, with functional units inside the MIPS. Instructions designed as low-cost are implemented using the `SUBLEQ` co-processor. Specifically, when such instructions are decoded, a sub-routine comprising a `SUBLEQ` program is invoked on the co-processor, performing the equivalent functionality as the low-cost-designated MIPS instruction. Our focus here is on `SUBLEQ`-based implementations for those MIPS instructions that are costly to implement in silicon, e.g., left and right shift (arithmetic and logical), and multiplication. In essence, our processor can be viewed as a *family* of processors, wherein the user pre-selects MIPS instructions as native or low-cost, thereby permitting easy performance/cost trade-offs.

A key aspect of our processor is its specification in `C` *software* written in a style that, when input to the LegUp open-source high-level synthesis tool from the University of Toronto, produces an optimized FPGA hardware implementation. The source code makes heavy use of predication to flatten the control-flow graph, simplifying the hardware's finite state machine (FSM) and reducing the number of cycles per instruction. Bitmasking is used in the `C` to hint the HLS and back-end RTL synthesis tools to shrink datapath widths. The processor's implementation is completely open source and available at: `https://github.com/Hara-Laboratory/Hirundo`. By specifying the processor at a high-level of abstraction in software, debugging and maintainability are improved, and it is straightforward to tailor to any application. We believe the embedded computing community will benefit from an open-source MIPS-ISA processor optimized for an HLS framework.

The contributions of this paper are:

- A MIPS-ISA processor enabling easy performance/cost trade-offs by way of it incorporating a `SUBLEQ` co-processor and associated sub-routines that permit execution of MIPS instructions.
- An open-source `C` specification of the processor that is intended/optimized for use with a state-of-the-art HLS tool.
- A complete `gcc`-based toolchain for targeting the proposed processor that produces a single hybrid MIPS/`SUBLEQ` binary.
- An experimental study comparing the area, cost and power consumption of FPGA implementations of a wide-variety of processor variants with an existing MIPS implementation described in Verilog RTL [3].

The remainder of this paper is organized as follows: Sec-

978-1-4673-8299-1/15 $31.00 © 2015 IEEE
DOI 10.1109/EUC.2015.23

tion II presents relevant background material and describes related work. The proposed architecture and its functionality is introduced in Section III. Section IV describes how MIPS instructions are translated into SUBLEQ subroutines. Section V describes our experiences with the HLS tool and details how we had to massage the C to get an optimized hardware implementation. The experimental study is described in Section VI. Conclusions and suggestions for future work appear in Section VII.

II. Background and Related Work

A. High-Level Synthesis

High-level synthesis refers to the automated synthesis of a hardware circuit from an untimed (clockless) software program. HLS is gaining traction recently as a design methodology for FPGAs, which can be used as configurable computing platforms to achieve higher throughput and energy efficiency than standard processors. In certain applications, recent work has demonstrated that the quality of circuit (area and performance) produced by HLS is close to that achieved with human-crafted hardware [4]. HLS is particularly attractive for applications where the specification/requirements change frequently, or in scenarios where there is demand for a variety of unique variants of a given circuit, each suited to particular application needs. It is the latter which motivated our use of HLS in this work: for the synthesis of a variety of processor variants with different area/speed trade-offs. In such cases, it is desirable to keep the functional specification in software where it can be easily modified, rather than in RTL, which is time-intensive and error prone to change. In this work, we use the LegUp open-source HLS tool which targets Altera FPGAs, and is built within the LLVM compiler framework [5].

B. Application-Specific Processors

Several commercial tools are available to generate custom processors (e.g., Xtensa [6]). With such tools, the user is able to generate a processor architecture customized to their application needs, as well as a compiler for the generated architecture. Thus, using such customized architectures requires that applications be re-compiled into the processor's specific (and proprietary) ISA, and necessitates that application source code be accessible.

Different application-specific processors [7], [8], [9] have been proposed based on such commercial tools. [8] describes a "trimmed" VLIW design methodology. The authors first created a baseline VLIW processor architecture by automatically deciding upon the various architectural components (e.g., number of registers, read/write ports, functional units, etc.) using a design space exploration algorithm based on ant colony optimization. Then, the base processor was optimized ("trimmed") by simulating a suite of application(s) on the fully programmable architecture and determining the unneeded components (such as unused interconnects, registers, multiplexers, and so on), which were then removed from the architecture. [7] presented a solution for constructing a processor core for a given application in C. As with the previous work, this used a baseline processor generated and optimized by commercial tools, and then the design was optimized for a set of applications. [9] similarly examined the area/performance benefits of tailoring an FPGA-based MIPS processor implementation for specific applications. The major limitation of such application-specific processors is that by tailoring them for a specific set of applications, they are no longer able to execute applications outside of those for which they were optimized. In our work, we overcome this limitation — our processor, while tailored for an application, is able to execute the full MIPS ISA.

C. Small-Scale Computers

Small computers with shrunken datapath widths have been proposed in prior work [10], [11], [12]. [10] presents an 8-bit processor, which emulates 32-bit ARM applications using its own instruction set that was designed to achieve a balance between area and performance. A similar approach was presented in [11] for FPGAs, comprising a 16-bit pipelined accumulator architecture with no register file. An extremely small version of these approaches was presented in [12], which implements the MIPS ISA and adopts a 2-bit serial data-path for its execution units. In order to perform 32-bit operations, each 32-bit operand is divided into sixteen 2-bit operands, and these operands are serially fed into the 2-bit wide execution units. In general, the weakness of such prior works is that they require many clock cycles to execute a single instruction; for example, [12] requires 23 cycles, on average, for each MIPS instruction.

Recently, one instruction set computers (OISC) have been proposed in [13], [14], [2] for applications with restrictive power/cost constraints. [13] describes an architecture that can natively process encrypted data in a cloud computing service, without ever sharing cryptography keys with a host machine. A SUBNEG-based OISC architecture has also been realized in post-silicon technology using carbon nanotubes [14]. A SUBLEQ co-processor was used in [2] to detect and recover from permanent faults in a host processor. While the co-processor could be re-purposed to create an application-specific processor with low area/power (as is done here), key differentiating features of our work are the area consumed by the host processor and the ease with which customization is achieved. [2] employs a hand-coded 5-stage pipelined host processor that consumes significant silicon area and requires low-level hardware design expertise to tailor to an application. In our work, on the other hand, the processor and co-processor are specified in C and can be tailored by anyone with software skills. Also, as will be demonstrated in the experimental study, our processor has a small area footprint, making it suitable for embedded applications. Lastly, we note that prior work on OISC architectures relied mainly on hand-coded applications — a significant impediment to the adoption of OISC is the lack of a compiler toolchain. In this work, we utilize a SUBLEQ co-processor in executing standard MIPS binaries.

III. Processor Architecture and Operation

In this section, we describe the proposed processor architecture and operation, and highlight some optimization techniques to improve its area and performance.

A. Usage Model

We envision the processor would be used as follows: 1) The user first compiles their application(s) and uses profiling to determine the instruction composition. 2) Based on the profiling results and the desired performance/area target, the user designates instructions to implement natively on the *host* processor (e.g., using a standard ALU) or with the low-cost SUBLEQ co-processor. In general, silicon area is reduced as a greater proportion of instructions are realized with the co-processor vs. on the host processor. Designating instructions as native/low-cost is straightforward, by changing a single C header file (to be discussed below). 3) The user compiles the processor using HLS to produce an RTL specification, synthesizable by existing back-end RTL synthesis tools.

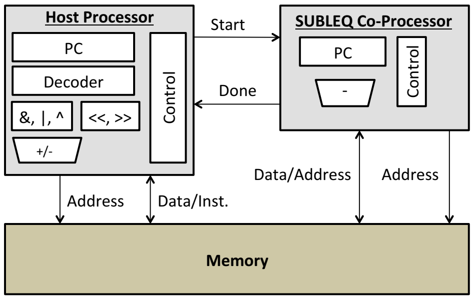


Fig. 1: Proposed processor architecture.

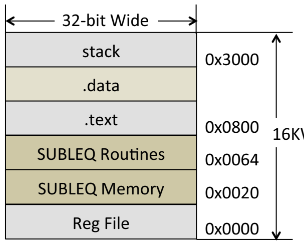


Fig. 2: Memory map of proposed architecture.

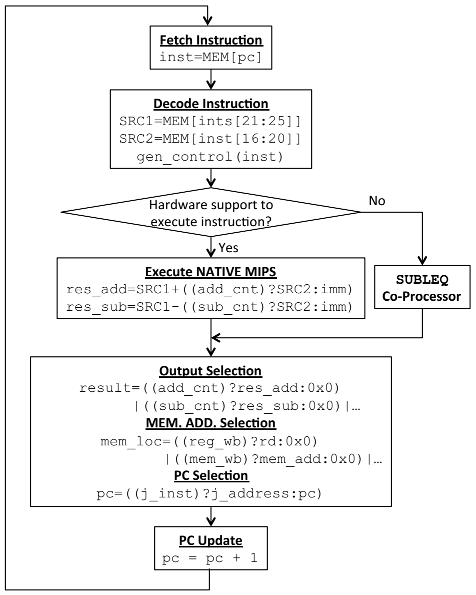

Fetch Instruction
inst=MEM[pc]
Decode Instruction
SRC1=MEM[ints[21:25]]
SRC2=MEM[inst[16:20]]
gen_control(inst)
Hardware support to execute instruction?
No
Yes
Execute NATIVE MIPS
res_add=SRC1+((add_cnt)?SRC2:imm)
res_sub=SRC1-((sub_cnt)?SRC2:imm)
SUBLEQ Co-Processor
Output Selection
result=((add_cnt)?res_add:0x0)
|((sub_cnt)?res_sub:0x0)|…
MEM. ADD. Selection
mem_loc=((reg_wb)?rd:0x0)
|((mem_wb)?mem_add:0x0)|…
PC Selection
pc=((j_inst)?j_address:pc)
PC Update
pc = pc + 1

Fig. 3: Instruction execution flow of the processor.

B. Architecture

Fig. 1 shows a block diagram of the processor connected to a memory. The figure illustrates the processor's internal functional blocks, comprising the expected logical, arithmetic, control and decoder blocks, along with a `SUBLEQ` co-processor. The co-processor is used to emulate MIPS instructions in the absence of a dedicated hardware resource. For example, in the absence of a hardware multiplier, the processor, on encountering a MIPS multiply (`mult`) instruction, invokes a sub-routine on the `SUBLEQ` co-processor that emulates the behavior of the `mult`.

The processor uses a common memory for instructions and data (von Neumann architecture) and, as a means for saving area, has no register file. Rather, the registers typically associated with the MIPS architecture reside in main memory. Fig. 2 shows the memory layout. Starting from address `0x0`, the lowest 32 locations of memory (i.e., `0x0–0x1F`) are used as a register file. The next two sets of memory locations are used for the `SUBLEQ` co-processor: the first set is used as scratch pad memory (described below); the next set holds the `SUBLEQ` sub-routines for emulating MIPS instructions. There is one sub-routine for each MIPS instruction. The rest of the memory is used for storing the program and data segments of the MIPS program. Note that we use a word-addressable memory to reduce the required number of memory accesses/ports. In a byte-addressable memory, one needs either to make multiple reads to access 32 bits (4 bytes) of data (takes multiple cycles), or alternately, one needs multiple memory ports that can be accessed concurrently. While we use the standard MIPS `gcc` compiler toolchain, we implemented a post-assembler script to convert the byte-addressable binary into a word-addressable binary. In this study, we use a 16KW (64KB) memory; however, any size of memory can be used.

C. Operation

The execution flow of the processor is shown in Fig. 3. First, an instruction is fetched from memory. The instruction is decoded, its type is ascertained and its operands are retrieved. If the instruction is to be implemented on the co-processor (see the decision block in Fig. 3), the base address of the sub-routine for the particular instruction type (which is hard-coded in the host processor) is passed to the co-processor and the instruction's operands are written to special memory locations (discussed below) so they can be accessed by the co-processor. Then, control passes to the co-processor, which is launched at the sub-routine address. The host processor waits until it receives a signal from the co-processor that the instruction has completed and the host processor can retrieve the results.

On the other hand, if the instruction is to be implemented on the host processor (natively), its execution proceeds in a traditional way using functional blocks within the host processor itself. Fig. 3 shows two representative examples for native execution on the host processor, namely, addition and subtraction. Following execution on either the host or co-processor, the appropriate result is selected from the co-processor or a relevant ALU. Likewise, the memory address at which to store the result is determined, as well as the next PC value, which may be the subsequent address or an address arising from a jump/branch. Observe that in the proposed processor, the host processor's program counter (PC) is always pointing to MIPS code; the co-processor's separate PC only points to `SUBLEQ` code.

IV. SUBLEQ Co-Processor

In this section, we describe how the host processor communicates with the co-processor, and also give some examples of `SUBLEQ` sub-routines that perform equivalent functionality to MIPS instructions. In general, MIPS instructions may have up to two source operands and produce a single computed result. Consequently, when the host processor decodes an instruction whose type is to be executed on the co-processor, it must "pass" the instruction operands to the co-processor and then retrieve the co-processor's result when the instruction has completed. This is done through specially designated memory locations which are pre-defined in advance. Specifically, the host processor moves instruction operands into specific

TABLE I: Memory information of SUBLEQ co-processor.

Memory locations	Purpose of memory
SRC1, SRC2	Source operands
DEST	Destination location
HI, LO	Special memory locations for multiplication
T0–T6	Scratchpad temporary mem locations
Z	Constant 0
INC, DEC	Constants (-1), $(+1)$
CW	Constant 32

memory locations whose addresses are hard-coded in SUBLEQ sub-routines. Similarly, SUBLEQ sub-routines place results into a specific location, which is hard-coded into the host processor. In essence, the pre-defined memory locations serve as a communication "bridge" between the host processor and the co-processor.

Besides the locations for operands and results, SUBLEQ sub-routines need access to a small scratchpad memory to store intermediate computed results, and we have found that 7 memory locations are sufficient for this purpose. In addition, we found that SUBLEQ sub-routines could be made more concise if several constant values were easily accessible, specifically, zero, $(+1)$ and (-1) and 32. $(+1)$ and (-1) are used by sub-routines to ease decrement and increment; 32 is the word width and is used by sub-routines that must walk through each bit of an operand (e.g., sub-routine for multiplication). We store these useful constants in 4 memory locations. Finally, the multiplication sub-routine required two additional locations to store the high- and low-order words of the computed 64-bit product. Such locations are also used by the host processor to handle multiply, even when the SUBLEQ processor is absent. Table I summarizes the special memory locations used by the SUBLEQ processor, and for communication between the host and co-processor.

Techniques for programming a SUBLEQ-based OISC architecture and some sub-routines for multiplication, division, and so on, have been presented in [15]. In the following, for the sake of brevity and owing to space limitations, we give two representative examples of SUBLEQ sub-routines for MIPS instructions. One is *unsigned add* (corresponds to MIPS instructions addu, addiu) and the other is for *logical shift left* (corresponds to the MIPS instruction sll). These two subroutines use some of the memory locations described in Table I.

We extended Mazonka's notation [15] for writing SUBLEQ sub-routines. By definition, an OISC architecture has solely one instruction type; consequently, we can omit specifying an opcode in SUBLEQ assembly. We use a semicolon and a newline to mark the end of each instruction. Operands are delimited by whitespace. A typical SUBLEQ sub-routine looks as follows:

```
@subroutine1 DEST, SRC1, SRC2
A B L;       // *B ← *B − *A
B DEST;      // *DEST ← *DEST − *B
L:Z Z End;   // *Z ← *Z − *Z (i.e., *Z ← 0)
```

where $*$ implies a pointer dereference. The first line is the header of the sub-routine. Each sub-routine has a header formatted like "@⟨subroutine name⟩ ⟨argument1⟩, ⟨argument2⟩, ..." and the arguments are fixed to "DEST, SRC1, SRC2" in the proposed architecture. Comments from "//" to end-of-line in each line are ignored. The first instruction in the sub-routine is interpreted as follows: 1) subtract the value in location A from the value in location B and store the result in location B; and 2) jump to location L if the computed value is non-positive, otherwise, proceed to the next instruction. When there are only two operands in one instruction, as in the second instruction, the implicit third operand is the address of the following instruction. In the third instruction, the address where first operand is stored is labeled as L. The third line is an example of a widely used idiom to clear the value at a memory address: "Z Z (...);". Formally, the parsing rules for sub-routines expressed in Extended Backus–Naur Form (EBNF) [16] are as follows:

$$\begin{aligned} \text{subroutine} &\Leftarrow \text{header } \{ \text{ instruction } \} \ ; \\ \text{header} &\Leftarrow \texttt{'@'} \text{ ident } [\text{ ident } \{ \ \texttt{','} \text{ ident } \} \ ] \ ; \\ \text{instruction} &\Leftarrow \text{operand operand } [\text{ operand }] \ \texttt{';'} \ ; \\ \text{operand} &\Leftarrow \{ \text{ ident } \texttt{':'} \ \} \text{ expr} \ ; \\ \text{expr} &\Leftarrow \text{number} \ | \ \text{ident} \ ; \end{aligned}$$

```
@addu DEST, SRC1, SRC2
// this subroutine does *DEST ← *SRC2 + *SRC1
L1:   SRC1     Z  L2;  // *Z ← *Z − *SRC1
L2:   SRC2     Z  L3;  // *Z ← *Z − *SRC2
L3:   DEST DEST  L4;  // *DEST ← 0
L4:      Z DEST  L5;  // *DEST ← *DEST − *Z
L5:      Z     Z End;  // *Z ← 0, and quit
```

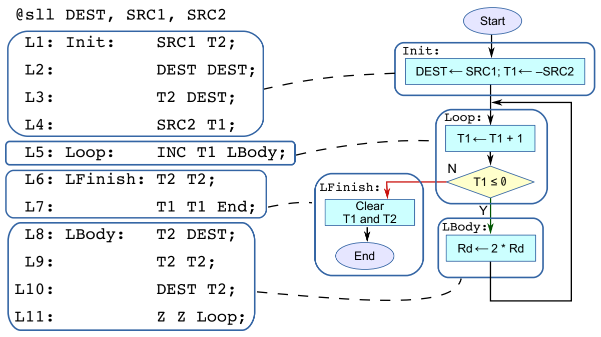
Fig. 4: SUBLEQ routine for unsigned addition.

a) Example 1: addu rd, rs, rt   Fig. 4 shows the SUBLEQ sub-routine that implements unsigned addition. The behavior of this MIPS instruction is to add two operands together (those in registers rs and rt) and write the result back to a destination register (rd). Prior to invoking the sub-routine, the host processor deposits the operands at locations SRC1 and SRC2. In the example, the first SUBLEQ instruction negates $*$SRC1 and stores it to memory location Z. The second instruction adds $*$SRC2 with $*$Z. The DEST location is shared by other sub-routines, and there exists the possibility of it containing a non-zero value. Thus, it is important to clear the location before storing the result, as is done in the third instruction. The fourth instruction updates the value at DEST. Finally, in the last instruction, the Z memory location is cleared for later use by other sub-routines and the SUBLEQ processor hands control back to the host processor (a constant End PC location is used to indicate sub-routine termination). The host processor retrieves the result from the DEST location.

b) Example 2: sll rd, rs, sa   Fig. 5 shows the SUBLEQ subroutine for logical left shift. The semantics of this MIPS instruction is to left-shift the value in register rs by an amount sa and deposit the result into the register rd. This example is more complicated than the previous one, as the SUBLEQ sub-routine uses more temporary locations than the previous example, and it contains a loop. The sub-routine repeatedly adds the value of rs to itself sa-times. As with the previous example, the operand and shift amount are stored into memory locations SRC1 and SRC2, respectively. For additional clarity, we also illustrate the sub-routine behavior in C:

```
*T2   = -(*SRC1);                       // L1
*DEST = -(*T2);                         // L2 to L3
*T1   = -(*SRC2);                       // L4
for ((*T1)++; *T1 <= 0; (*T1)++) {  // L5
    *DEST -=  *T2;                      // L8
    *T2    = -(*DEST);                  // L9 to L10
}
*T1 = *T2 = 0;                          // L6 to L7
```

Temporary registers (e.g., T1 and T2) are initially cleared to zero. The sub-routine contains four chunks: 1) initialization, 2) counter update and loop condition, 3) loop body, and 4) the



```
@sll DEST, SRC1, SRC2
 L1: Init:     SRC1 T2;
 L2:           DEST DEST;
 L3:           T2 DEST;
 L4:           SRC2 T1;
 L5: Loop:     INC T1 LBody;
 L6: LFinish:  T2 T2;
 L7:           T1 T1 End;
 L8: LBody:    T2 DEST;
 L9:           T2 T2;
L10:           DEST T2;
L11:           Z Z Loop;
```

Fig. 5: Implemantation in SUBLEQ of instruction "sll rd, rs, sa", which is equivalent to rd = rs << sa.

ending. 1) The first three instructions (L1–L3) copy the value from location SRC1 to DEST using the temporary location T2. $\star$T2 equals ($-\star$DEST) after executing these instructions. The instruction on L4 deposits the negated shift amount into location T1. 2) We combine updating the counter $\star$T1 and checking the loop condition into one instruction at L5. The instruction "INC T1 Lbody;" increments the counter $\star$T1 by one and jumps to the loop body if the counter is non-positive. Note that the counter has been initialized with a negative value ($-$sa); therefore, the loop body will be executed sa times. 3) The loop body relies on an invariant that ($\star$T2) $= -$($\star$DEST). The instruction at L8 subtracts $\star$T2 from $\star$DEST. This instruction therefore doubles the value in DEST (i.e., left shift) because of the loop invariant. The ninth instruction clears location T2, and the tenth instruction moves ($-\star$DEST) into T2, maintaining the loop invariant. The last instruction in the loop body clears location Z; this subtraction has no effect, and its result is always zero, causing a jump to L5. 4) Finally, all the used temporary registers are cleared to zero by sixth and seventh instructions before returning control to the host processor.

Available SUBLEQ Sub-routines

Despite that the SUBLEQ OISC architecture is Turing complete and therefore able to perform *any* computation [2], in some cases, the SUBLEQ equivalents of certain MIPS instructions are not performance efficient, requiring a large number of SUBLEQ operations. Thus, in this paper, we provide knobs that permit a subset of MIPS instructions to be designated for the co-processor: those MIPS instructions that require significant area to realize in the host processor's ALU (and would provide a significant area reduction if eliminated), and/or those instructions that require relatively few SUBLEQ instructions to emulate on the co-processor. The following classes of instructions can be implemented with SUBLEQ subroutines:

- **Multiplication:** both multiplication (i.e., mult, and multu) and move operations related to multiplication (i.e., mflo, mfhi, mtlo, and mthi) are replaced with SUBLEQ sub-routines.
- **Subtraction:** both sub, and subu of the MIPS ISA are replaced with SUBLEQ sub-routines.
- **Shift:** all three shift operations of MIPS ISA (i.e., sll, sllv, srl, srlv, sra, and srav) are replaced with three SUBLEQ routines which are shift-left-logical, shift-right-logical, and shift-right-arithmetic.
- **Set-less-than:** set-less-than is commonly used for processing the control flow of the program. Four instructions slt, slti, sltu, and sltiu are replaced with SUBLEQ sub-routines.
- **Addition:** all MIPS instructions which use the adder unit of the host processor such as, add, addu, addi, and addiu.

When one of the above instruction classes is designated to run on the co-processor, the necessary hardware to realize that class in the host processor is eliminated, thereby reducing the area of the host processor.

A configuration file is used to generate different variants of the processor, which manages the hardware resources of the host processor and also generates control for the co-processor during HLS. With 5 classes of instructions that can be implemented on the host or co-processor, the user is able to generate $2^5 = 32$ unique processors, by way of minor changes to the configuration file.

V. C Specification for HLS

In this section, we highlight a few features of our C implementation that we found to be necessary to produce optimized hardware. A key weakness of modern HLS tools is the notion of *syntactic variance*: the circuit generated by HLS depends significantly on the style of the input program and also the constraints provided to the HLS tool. Recent work has shown that a specific coding style may be necessary to produce good-quality results [17].

Our implementation has an outer while loop that continues to execute until the program terminates. Each iteration of the loop fetches and executes a single MIPS instruction.

A. Executing an Instruction

We declare a Boolean variable for each instruction type and initialize the variable to true if the variable corresponds to the current instruction being executed.

```
     ...
1:   bool R_TYPE    = (opcode == 0x00);
2:   bool ADDU_COND = (R_TYPE & (funct == 0x21));
3:   bool OR_COND   = (R_TYPE & (funct == 0x25));
     ...
4:   bool BNE_COND  = (opcode == 0x05);
     ...
```

Line 1 above checks if the instruction is a register-type instruction. Lines 2 and 3 show the variable declarations for unsigned addition (addu) and bitwise OR (or) — these query the funct 6-bit field in register-type MIPS instructions. Line 4 shows the variable declaration for branch-if-not-equal (bne).

At a high level, MIPS instructions take one or more of the following three actions: 1) perform a logical/arithmetic operation on two register operands, or a register and an immediate value, and then deposit the result to a register; 2) perform a write to memory; or 3) jump/branch to a location (possibly conditional). Our initial C implementation made heavy use of conditionals (if-else) to perform *only* the work needed for the particular instruction being executed. While it produced functionally correct results, this coding style produced a complicated control-flow graph (CFG), and a correspondingly complicated FSM in the HLS-generated hardware. Moreover, we originally had many locations in our implementation where we read/wrote from/to memory. For example, we had separate locations in our source code for memory writes for sw (store word) and sb (store byte). This approach, however, produced large multiplexers on the RAM inputs in the hardware implementation, bloating area.

We found that superior HLS results were produced by the following approach for the various actions an instruction may take.

1) Arithmetic/Logical: We compute *all* possible arithmetic/logical results irregardless of the type of the current instruction. We then select the specific relevant result for the current instruction, ignoring all other computed results. A representative code snippet is as follows:

```
   ...
1:  unsigned int RES_AND  = SRC1 & LOGIC_IMM_INP;
2:  unsigned int RES_OR   = SRC1 | LOGIC_IMM_INP;
3:  int          RES_SUBU = SRC1 - SRC2;
    ...
4:  RESULT = ((AND_COND) ? RES_AND  : 0x0) //AND
           | ((OR_COND)  ? RES_OR   : 0x0) //OR
           | ((SUB_COND) ? RES_SUBU : 0x0) //SUBU
           | ...
    ...
```

Lines 1–3 above perform the computations for logical AND, OR and unsigned subtraction. In lines 1 and 2, the logical operation is performed between a source register `SRC1` and an immediate operand `LOGIC_IMM_INP`. Line 4 selects a particular pre-computed result, based on the opcode of the current instruction. Note in line 4 the use of logical OR: a single ternary (`<> ? <> : <>`) operator condition evaluates to `true` and the particular result is OR'ed with zeroes (the evaluated result of all-but-one of the ternary operators).

2) Writing Back to Memory: We similarly compute *all* possible write-back address locations and then select the correct location for the current instruction. If the current instruction does not perform a memory write, we default to a dummy unused write-back address. An example code snippet:

```
    ...
1:  bool WB_RD = (SLL_COND | AND_COND | ...
2:  bool WB_RT = (SLTI_COND | SLTIU_COND | ...
    ...
3:  WB_LOC = ((WB_RD)? rd : 0x0)
           | ((WB_RT)? rt : 0x0)
           | ...
    ...
```

Lines 1–2 set Boolean variables according to whether a write will occur to the register specified in the `RD` or `RT` fields of the instruction word — this depends on the type of instruction being executed. Line 3 selects the actual write-back address.

3) Jump/Branch: We likewise compute *all* possible jump/branch locations and relevant branching conditions, and as above, we then select the correct destination/condition for the current instruction. If the current instruction is not a jump or branch, the destination will be the program counter.

To be sure, the above approach performs redundant work for each instruction, however, with this approach, the control-flow graph for the code to execute an instruction consists of a single basic block — straight-line code with no branches. This simplified the FSM of the HLS-generated HW.

B. *SUBLEQ Co-Processor*

The co-processor is specified in 10 lines of `C` code:

```
 1: while (PC != End) {
 2:   unsigned short a = get_value (PC);
 3:   unsigned short b = get_value (PC + 0x1);
 4:   unsigned short c = get_value (PC + 0x2);

 5:   signed int src1 = get_value (a);
 6:   signed int src2 = get_value (b);
 7:   src2 = src2 - src1;
 8:   write_value (src2, b);
 9:   PC = 0x3FF & ((src2 > 0) ? PC + 0x3 : c);
10: }
```

where the outer `while` loop continues to execute a sub-routine until a sentinel program counter (`PC`) value is reached (`End`), and `get_value`/`write_value` are memory read/write. The `SUBLEQ` routines for MIPS instructions are located in the lower part of the address space (below address `End`). Lines 2–4 retrieve the operands, `a`, `b` and `c` from memory. Lines 5–6 dereference pointers `a` and `b`. The subtract and store occur in lines 7 and 8, respectively. The new `PC` is computed in line 9. Masking with `0x3FF` (10 bits) in line 9 reduced the datapath width for the addition and select in the generated hardware.

Regarding the bitmasking in line 9 of the code example above, by zeroing all but the lower bits of a computed value, the HLS and back-end RTL synthesis tools are able to infer that all high-order bits are logic-`0` and therefore generate no hardware to produce such bits. While we cannot apply bitmasking on register-data calculations (these must be 32-bits), we did apply such bitmasking throughout our `C` implementation on addressing-related computations and computations involving immediate values (these are 16-bits wide in the MIPS ISA).

VI. Experimental Study

In this section, we present the experimental evaluation of our proposed processor. First, the experimental setup is described and then experimental results are discussed for the following metrics: 1) area, 2) power, 3) wall-clock time, and 4) area-delay product. We report power (as opposed to energy) owing to its importance in embedded/IoT scenarios, where devices are supplied by sources with limited peak power, such solar (e.g. [18]) or the EM environment.

A. *Experimental Setup*

The experimental setup and toolchain is shown in Fig. 6, leveraging in-house, open-source, and commercial tools. Fig. 6 shows that an in-house assembler① (written in `Haskell`) is used to generate `SUBLEQ` sub-routine libraries, which are later linked by `elf2mem`④ with the MIPS binary generated by GCC-4.1.1③. We evaluate processor variants using 12 benchmark programs collected from the SNU real-time [19] and CHStone [20] benchmark suites. Benchmarks were compiled with `-O2` optimization. `SUBLEQ` sub-routines were verified by an in-house `SUBLEQ` sub-routine verifier②. The generated test vectors (the combined MIPS/`SUBLEQ` binary executable), the `C` description of the processor and the configuration file designating which instruction types are native/low-cost (`config.h`) are provided as inputs to the LegUp HLS tool⑤, generating RTL code for the processor.

The processor's RTL is simulated using ModelSim⑥ to verify functional correctness and gather the total number of execution cycles required to execute each benchmark application. The RTL is also synthesized, placed and routed in Altera's 90 nm Cyclone II [21] FPGA using Quartus II ver. 11.1⑦, allowing us to extract area and clock frequency ($f_{\max}$) from the post-routing implementation⑧. Combining the $f_{\max}$ with the cycle count data allows us to compute wall-clock time for each benchmark: (# of cycles) $\times\ 1/f_{\max}$. Altera tools were used to generate a post-routing gate-level structural netlist and delay file (`.sdf`) enabling us to execute a full-delay gate-level simulation with ModelSim⑨, producing a switching-activity (`.vcd`) file. The activity data and routed design were input to

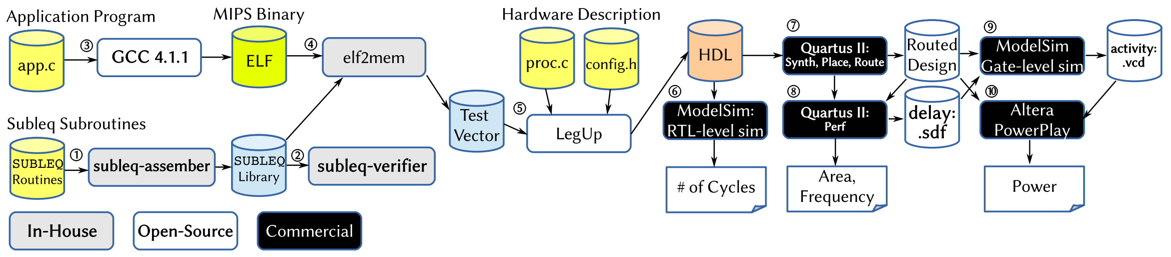


Fig. 6: Experimental setup for the evaluation.

Altera's power analyzer⑩, yielding a power estimate reflecting post-routed delays and interconnect capacitances.

We consider five different processor variants:

1) **HOST ONLY**: All instruction classes execute on the host; there is no SUBLEQ co-processor. For this processor, we synthesized it in two ways: 1) using the hardened ASIC-like multiplier blocks on the Cyclone II, and 2) without using the hardened multipliers by directing the Altera RTL synthesis tools to implement multipliers using the FPGA's soft logic: look-up-tables (LUTs). We include the soft logic multiplier implementation because some low-cost/low-power FPGAs for embedded applications (e.g., [22]) do not contain hardened multipliers.
2) **M-SUBLEQ**: Only multiplication instructions run on the SUBLEQ co-processor.
3) **MS+-SUBLEQ**: Multiplication, shift, subtraction, and set-less-than instructions run on the SUBLEQ co-processor.
4) **ALL-SUBLEQ**: All the instructions discussed in Section IV are run on the SUBLEQ co-processor, and the few instructions remaining are executed on the host processor.
5) **TIGER**: As a point of comparison, we also include results for the TIGER MIPS processor from the University of Cambridge [3]. TIGER is a *hand-designed* Verilog RTL implementation of a MIPS processor with a 5-stage pipeline and separate instruction and data caches (9KB each). With respect to comparing results between TIGER and our processors, the reader should bear in mind that, as opposed to TIGER, our processor is automatically synthesized from C (making it straightforward to modify) and is intended for low-cost/low-power scenarios where performance can be sacrificed to some extent. As with #1 above, we synthesized TIGER two ways: with and without using the hardened multipliers on Cyclone II.

B. Experimental Results

1) Profiling the MIPS Instruction Mix: We executed each benchmark in a cycle-accurate MIPS simulator to profile the dynamic instruction mix. Such profiling data is valuable for interpreting the results later in this section, and for selecting instruction classes to execute on the SUBLEQ co-processor for a given benchmark. In general, instruction types that execute infrequently are potentially good candidates to move to the co-processor, trimming area from the host processor with moderate performance consequences. Profiling results are shown in Fig. 7, where instruction types are partitioned into 6 categories. Starting at the bottom of each bar (multiplier),

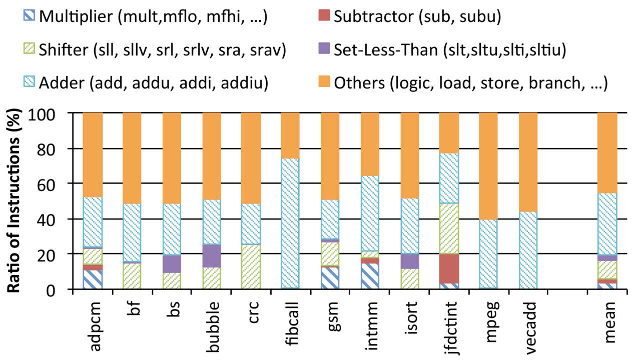


Fig. 7: MIPS instruction mix profile.

observe that only 4 benchmarks require multiplication (which includes the following six MIPS instructions mult, multu, mflo, mfhi, mtlo, and mthi). Across all benchmarks, 3.6% of instructions, on average, reside in this category. Shift instructions comprise 10.7% of all instructions executed, on average. Subtract and set-less-than instructions represent 1.9% and 3% of all executed instructions, respectively. The lion's share of all instructions lie in the adder and "other" category. Over 35% of instructions, on average, require addition. The "other" category includes the logical operations (*logical and, or, nor,* and *xor*), loads/stores, and control-flow instructions (branch/jump).

2) Area and $f_{\max}$: Area and $f_{\max}$ results for all processor variants are given in Table II. We report the number of Cyclone II logic elements (LEs), each of which comprises a 4-input LUT/flip-flop pair, as well as the number of multiplier blocks (9×9 hardened multipliers). $f_{\max}$ results reflect post-routing delays. The columns of the table correspond to the processor variants mentioned above. Note that there are two columns for the HOST ONLY and TIGER processors for the implementations with and without using the hard multipliers, respectively. The TIGER distribution from the University of Cambridge is a complete system that operates on Altera's Cyclone II-based DE2 board, with on-chip cache backed by on-board SDRAM memory, integrated debugging and other surrounding logic. To create a more apples-to-apples comparison between TIGER and the proposed processors, the reported area/power/performance results reflect solely the processors, and do not include the logic in other parts of the system. Also, since the benchmark programs did not require division, neither our processors nor TIGER include divider units[1].

The first row of Table II shows the number of Cyclone II

[1]We modified TIGER's Verilog RTL to remove the divider units.

TABLE II: Synthesized area and frequency.

	HOST ONLY		w/ SUBLEQ Co-Processor			TIGER	
	Hard Mult.	**Soft Mult.**	**M-SUBLEQ**	**MS+-SUBLEQ**	**ALL-SUBLEQ**	**Hard Mult.**	**Soft Mult.**
LEs	1821	2185	1600	1232	1141	4232	5709
Ratio	1.00	1.19	0.88	0.68	0.63	2.32	3.13
Ratio	0.83	1.00	0.73	0.56	0.52	1.94	2.61
# of Multipliers	6	0	0	0	0	8	0
Frequency (MHz)	150.7	149.2	154.6	169.9	176.8	89.9	87.9
Ratio	1.00	0.99	1.02	1.13	1.17	0.60	0.58
Ratio	1.01	1.00	1.03	1.14	1.18	0.60	0.59

LEs. The two subsequent rows show the ratios of LE usage relative to the HOST ONLY processor with and without the hard multipliers, respectively. Columns 4–6 contain results for processors with the SUBLEQ co-processor. Observe that as successively more instruction classes are handled by the co-processor, significant area reductions are achieved. Relative to the HOST ONLY processor with hardened multipliers, a 37% drop in LE usage is realized in the ALL-SUBLEQ scenario. When soft LE-based multipliers are used, a nearly 2× reduction in area is seen for the ALL-SUBLEQ processor vs. HOST ONLY. The ALL-SUBLEQ processor requires just over 1000 LEs. Row 4 of the table shows the hardened multiplier usage. It is worth reinforcing that, since our processor is described in C and synthesized with HLS, all variants, which exhibit the wide range of areas shown in Table II, were realized by software changes alone.

The two right-most columns of the table give the results for TIGER, which requires ∼2–3× more LEs than the HOST ONLY processor, depending on the style of multiplier implementation. In comparison with the ALL-SUBLEQ processor, TIGER requires ∼4–5× more LEs (cf. the ALL-SUBLEQ column and the two right-most columns of the table). We believe that, owing to its low resource requirements, our processor will be useful in area-constrained scenarios requiring a processor that can execute a standard MIPS binary produced via a standard toolchain.

The last rows of Table II pertain to $f_{\max}$, and follow an analogous format to the LE rows of the table. Observe that as more instruction classes are migrated to the co-processor, $f_{\max}$ increases by up to ∼17–18% as the host processor's datapath and control hardware is reduced. TIGER operates at a significantly lower $f_{\max}$ than the proposed processor: ∼40% lower than HOST ONLY. TIGER's low $f_{\max}$ is a conequence of its considerably more complicated architecture, control and data-forwarding structures.

3) Power Consumption: Table III gives power consumption results for two benchmarks, gsm and isort (insertion sort)[2]. We selected these two benchmarks for detailed power analysis because gsm executes a diverse set of instruction types, while isort requires a limited set of instruction types. Looking first at the power consumption of the proposed processors, observe that the lowest power consumption for these two benchmarks was with the M-SUBLEQ processor, where only multiply instructions are handled by the co-processor. The insertion sort benchmark did not require multiplication and hence, for this benchmark the co-processor is never invoked in the M-SUBLEQ configuration and the multiplier functional unit is eliminated from the host processor, leading to lower overall power.

To analyze the power consumption of TIGER, we simulated the complete TIGER system (with cache) to gather detailed post-routing switching activity data. We then scaled up the activities based on the fraction of cycles in which the TIGER processor was stalled/idle due to cache misses, producing a more fair comparison with the proposed processors (which use on-FPGA memory and do not experience cache misses). Overall, TIGER requires between ∼2.5–4× more power than the proposed processor, depending on the configuration chosen. With respect to static (leakage) power, the Altera power analysis tools do not report leakage in the used vs. unused portion of the FPGA. However, leakage current generally tracks with total transistor width (silicon area) and we therefore expect TIGER to exhibit ∼4–5× more static power than the ALL-SUBLEQ processor.

4) Wall-Clock Time: Although the primary motivation for the proposed processors is to achieve low-cost and power, for completeness, we assessed performance as well. Table IV shows the wall-clock time (in μs) for different variants of processor. Observe that, relative to HOST ONLY, the co-processor-based configurations require 2.4-8.4× more wall-clock time, on average, owing to the number of SUBLEQ instructions that need to be executed to emulate MIPS instructions. However, digging further into the data, we see that for the M-SUBLEQ configuration, 8/12 benchmarks have superior wall-clock time vs. HOST ONLY. When multiplications are handled by the co-processor, the need for the host processor to handle wide 64-bit products is eliminated, reducing the number of memory access points in the host, decreasing its latency. As long as such latency reduction is not offset by lengthy multiply sub-routines on the co-processor, a win in wall-clock time is achieved. Additional wall-clock time reductions are seen for two benchmarks (fibcall, and vecadd) in the MS+-SUBLEQ configuration.

As expected, the hand-designed pipelined TIGER processor exhibits the best wall-clock time, ∼4× lower than HOST ONLY, on average. The primary reason for this is TIGER's use of a register file and separate instruction/data caches[3]. In our processors, instruction fetch, register operations, and loads/stores are directed to a single main memory, which, when implemented using on-FPGA memory, is dual-ported with single-cycle access. The dual-port limitation implies that several cycles are required to execute a single instruction in our processor; whereas, TIGER ideally issues a new instruction each cycle. We chose the memory style of our processors to target the low-cost/power embedded space. An interesting direction for future work is to assess our processors with cache and register file.

5) Area-Delay product: Table V shows area-delay product results, where area is measured in LEs and we have estimated the area of a hard-multiplier tile in Cyclone II (which contains two 9×9 multipliers) to be equal to two LABS = 32 LEs[4] [23].

[2]We focused on two benchmarks owing to the considerable run-time required for full-delay post-layout simulation.

[3]TIGER's cycle count data reflects cycles spent in useful work (not stall cycles due to cache misses).

[4]A LAB (logic array block) in Cyclone II contains 16 4-LUT/FF pairs.

TABLE III: Dynamic power consumption for two benchmarks (mW).

Benchmark	HOST ONLY		w/ SUBLEQ Co-Processor			TIGER	
	Hard Mult.	**Soft Mult.**	**M-SUBLEQ**	**MS+-SUBLEQ**	**ALL-SUBLEQ**	**Hard Mult.**	**Soft Mult.**
gsm	26.2	32.2	26.4	23.9	27.0	60.6	85.3
isort	18.4	21.4	16.2	20.7	23.4	67.6	87.5
geomean	22.0	26.2	20.7	22.3	25.1	64.0	86.4
Ratio	1.00	1.19	0.94	1.01	1.14	2.92	3.93
Ratio	0.84	1.00	0.79	0.85	0.96	2.44	3.30

TABLE IV: Wall-clock time (μs).

Benchmark	HOST ONLY		w/ SUBLEQ Co-Processor			TIGER	
	Hard Mult.	**Soft Mult.**	**M-SUBLEQ**	**MS+-SUBLEQ**	**ALL-SUBLEQ**	**Hard Mult.**	**Soft Mult.**
adpcm	5537.3	5595.7	101366.1	125499.0	126028.1	1702.7	1748.7
bf	35484.2	35858.8	29971.6	371667.6	396062.3	7803.0	8013.8
bs	4.3	4.4	3.7	40.9	43.6	0.9	0.9
bubble	106513.6	107638.2	100512.9	327637.5	405918.3	28132.7	28893.0
crc	1544.2	1560.5	1304.4	20855.6	21248.7	419.4	430.8
fibcall	9.1	9.2	7.7	7.6	29.6	2.1	2.1
gsm	1365.7	1380.1	34346.1	44063.1	43412.8	430.3	442.0
intmm	31624.6	31958.5	915082.7	879034.3	891176.2	8217.7	8439.8
isort	47.7	48.2	40.4	122.0	167.7	11.7	12.0
jfdctint	218.8	221.2	2085.2	4000.1	4061.5	56.1	57.7
mpeg	579.5	585.6	489.5	519.4	1257.3	160.2	164.5
vecadd	478.8	483.8	404.5	371.8	1061.8	100.6	103.3
geomean	858.2	867.3	2064.5	5167.9	7204.9	217.3	223.1
Ratio	1.00	1.01	2.41	6.02	8.40	0.25	0.26
Ratio	0.99	1.00	2.38	5.96	8.31	0.25	0.26

Similar to the wall-clock time results, observe that most of the benchmarks exhibit their best area-delay product in one or more of the co-processor configurations (vs. in HOST ONLY), although the average results do not reflect this trend, because of the large wall-clock times seen for several benchmarks when the co-processor is used.

The TIGER processor, because of its low wall-clock time (discussed above) achieves the best area-delay product overall, on average. However, we are nevertheless encouraged that some benchmarks (e.g., `fibcall`) achieve close-to or better area-delay product than TIGER (e.g., with soft multipliers), especially considering our processors are synthesized from `C` using HLS, and TIGER is hand-designed RTL. We again underscore that the proposed processors are intended for low-cost/power embedded applications; the performance of our processors can be improved (at the expense of area) by incorporating memory structures similar to TIGER (e.g., a register file).

6) Application-Specific Processor Generation: From the results above, it is apparent that no single processor offers the best area-delay product for all applications, rather it depends on the operations in the applications. In this section, we have selected three applications which exhibit the best area-delay product in different processors. Fig. 8 gives the normalized area-delay product for `jfdctint`, `crc`, and `mpeg` for the following processor configurations: 1) HOST ONLY (w/ soft mult.), 2) M-SUBLEQ, MS+-SUBLEQ, ALL-SUBLEQ, and TIGER (w/ soft mult.). Note that the area-delay product for ALL-SUBLEQ is high due to the large wall-clock time, and, although this configuration is attractive because of its low area, there is no application found which performs better in this configuration from the area-delay product angle.

In Fig. 8 the area-delay product values are normalized to the `HOST ONLY` results. The leftmost benchmark, `jfdctint`, has a diverse set of instruction types, including multiplication operations. For this benchmark, when multiplication operations are moved into the co-processor, the wall-clock time increases significantly, overwhelming the gain from the area reduction. The second application, `crc`, has no multiplication operations,

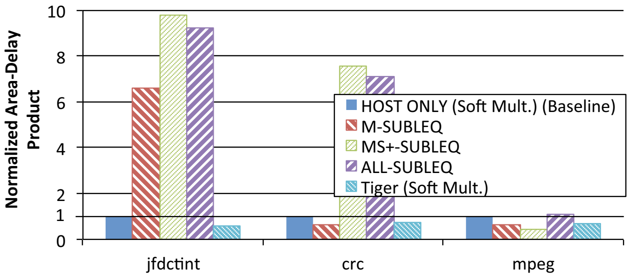


Fig. 8: Area-delay product for three applications (normalized to HOST ONLY).

and by moving multiplication into the co-processor, area as well as wall-clock time are reduced — area-delay product is reduced by 40%. However, when more instruction types are moved into the co-processor, the wall-clock time hit overwhelms the gain from the area reduction. Since `crc` contains shift operations, a considerable wall-clock time increase is observed when shift is done in the co-processor. The last benchmark, `mpeg`, illustrates effectiveness of the MS+-SUBLEQ configuration, which offers a 50% reduction in area-delay product vs. the baseline. Surprisingly, for this benchmark, area-delay product in MS+-SUBLEQ is even superior to that of the hand-designed TIGER processor.

Although different processor configurations perform best for certain applications, a key feature of the proposed processors is that they are all able to execute the full MIPS ISA. This is as opposed to prior work in [7], [8], [9], where the application-specific processors were only able to execute the applications for which they were optimized.

VII. Conclusions and Future Work

In this paper, we have proposed a processor architecture that executes standard MIPS-ISA binaries and permits tradeoffs between area/performance/power and is straightforward to tailor to specific applications (while maintaining full support for the MIPS ISA). The unique architecture incorporates a

TABLE V: Area-delay product (#LEs/1000 × ms).

Benchmark	HOST ONLY		w/ SUBLEQ Co-Processor			TIGER	
	Hard Mult.	**Soft Mult.**	**M-**SUBLEQ	**MS+-**SUBLEQ	**ALL-**SUBLEQ	**Hard Mult.**	**Soft Mult.**
adpcm	10.3	12.2	162.2	154.6	143.8	7.3	10.0
bf	66.3	78.4	48.0	457.9	451.9	33.5	45.8
bs*	8.1	9.6	6.0	50.4	49.7	4.0	5.4
bubble	199.1	235.2	160.8	403.6	463.2	120.9	165.0
crc	2.9	3.4	2.1	25.7	24.2	1.8	2.5
fibcall*	17.0	20.0	12.3	9.4	33.8	8.8	12.0
gsm	2.6	3.0	55.0	54.3	49.5	1.8	2.5
intmm	59.1	69.8	1464.1	1083.0	1016.8	35.3	48.2
isort*	89.2	105.4	64.6	150.2	191.4	50.1	68.3
jfdctint	0.4	0.5	3.3	4.9	4.6	0.2	0.3
mpeg	1.1	1.3	0.8	0.6	1.4	0.7	0.9
vecadd	0.9	1.1	0.6	0.5	1.2	0.4	0.6
geomean	1.6	1.9	3.3	6.4	8.2	0.9	1.3
Ratio	1.00	1.18	2.06	3.97	5.13	0.53	0.73
Ratio	0.85	1.00	1.74	3.36	4.34	0.45	0.62

*Values are in milli (10^{-3}) unit.

SUBLEQ OISC co-processor that can be used to emulate the functionality of different classes of MIPS instructions, allowing the main processor's datapath/control logic to be reduced, saving area. The processor is described in C and synthesized to hardware with HLS, making it easy to modify and extend — a feature we believe will keenly interest members of the embedded computing community, enabling new research on lightweight embedded processors. Results show the processor uses 2.5–4× less area than a hand-designed Verilog RTL pipelined MIPS processor, and 4–5× less power, depending on the particular processor configuration used. For certain applications, the proposed processor also achieves better area-delay product than the pipelined MIPS.

Directions for future work include studying how the processor's performance changes when different memory architectures are used (e.g., register files and caches), as well as considering standard-cell implementations, and exploring the impact of HLS constraints (e.g., loop pipelining) on the implementation results. We also plan to explore power-optimization strategies, such as clock gating the co-processor or host processor accordingly when idle. The proposed processor, benchmarks and associated toolchain are open-source and available at: `https://github.com/Hara-Laboratory/Hirundo`.

ACKNOWLEDGEMENT

This work was partially supported by KAKENHI 15H02679 and the Japan Society for the Promotion of Science (JSPS).

REFERENCES

[1] A. Canis *et al.*, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, 2013.

[2] A. Rajendiran *et al.*, "Reliable computing with ultra-reduced instruction set co-processors," in *Proc. of DAC*, pp. 697–702, 2012.

[3] University of Cambridge, `http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html`, *The Tiger "MIPS" processor.*, 2010.

[4] J. Cong *et al.*, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. on CAD*, vol. 30, no. 4, pp. 473–491, 2011.

[5] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. of CGO*, pp. 75–88, 2004.

[6] R. E. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, 2000.

[7] J. Trajkovic and D. Gajski, "Custom processor core construction from C code," in *Proc. of SASP*, pp. 1–6, 2008.

[8] J. Matai *et al.*, "Trimmed VLIW: Moving application specific processors towards high level synthesis," in *Proc. of ESLsyn*, pp. 11–16, 2012.

[9] P. Yiannacouras, J. G. Steffan, and J. Rose, "Exploration and customization of FPGA-based soft processors," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 266–277, 2007.

[10] Y. Hara-Azumi, M. Kunimoto, and Y. Nakashima, "Emulator-oriented tiny processors for unreliable post-silicon devices: A case study," in *Proc. of ASP-DAC*, pp. 85–90, 2014.

[11] M. Schoeberl, "Leros: A tiny microcontroller for FPGAs," in *Proc. of FPL*, pp. 10–14, 2011.

[12] H. Nakatsuka *et al.*, "Ultrasmall: The smallest mips soft processor," in *Proc. of FPL*, pp. 1–4, 2014.

[13] N. Tsoutsos and M. Maniatakos, "HEROIC: Homomorphically encrypted one instruction computer," in *Proc. of DATE*, pp. 1–6, 2014.

[14] M. M. Shulaker *et al.*, "Carbon nanotube computer," *Nature*, vol. 501, pp. 526–530, Sept. 2013.

[15] O. Mazonka and A. Kolodin, "A simple multi-processor computer based on subleq." preprint on arXiv: CoRR/1106.2593, May 2011.

[16] ISO/IEC, "Information technology — Syntactic metalanguage — Extended BNF," ISO/IEC 14977:1996, 1996.

[17] J. Matai *et al.*, "Designing a hardware in the loop wireless digital channel emulator for software defined radio," in *Proc. of FPT*, pp. 206–214, 2012.

[18] W. Lim *et al.*, "Batteryless sub-nw Cortex-M0+ processor with dynamic leakage-suppression logic," in *Proc. of ISSCC*, pp. 146–147, 2015.

[19] "SNU real-time benchmarks." `http://archi.snu.ac.kr/realtime/benchmark`.

[20] Y. Hara *et al.*, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

[21] Altera, Corp., *Cyclone-II FPGA family datasheet*, 2015.

[22] Lattice Semiconductor, *iCE40 FPGA family datasheet*, 2015.

[23] Altera, Corp., *Personal communication*, 2015.