This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier xxx

A Multi-One Instruction Set Computer for Microcontroller Applications

MARCO CREPALDI[1], (Member, IEEE), ANDREA MERELLO[1,2], MIRCO DI SALVO[1,2]
[1]Istituto Italiano di Tecnologia, Electronic Design Laboratory (EDL), Via Enrico Melen 83, 16152 Genova, Italy.
[2]*Equal Contribution.*

Corresponding author: Marco Crepaldi (e-mail: marco.crepaldi@iit.it).

ABSTRACT This work presents a simple integer-only instruction set architecture and microarchitecture derived from One Instruction Set Computers (OISCs) and embedding *multiple* execution modes (*m*OISC), capable of running at a reasonable performance level to enable basic usability in microcontroller applications. The purpose of *m*OISC is to enable simple data transfer tasks and run small programs while maintaining ultimate simplicity. We present the internal organization for a computer architecture including an 8 bit I/O register, and 64 kB central Random Access Memory (RAM), organized in two-bytes words. The processor can run code generated assuming an OISC or a Complex Instruction Set Computer (CISC) scheme (op-code based), depending on the programmer's demands and based on the initial setting of a register during start-up. To enable practical applications and demonstrate successful exploitation of *m*OISC in view of integration in a compiler back-end, we designed a custom Proof-of-Concept (PoC) software design toolchain based on LLVM and `clang`. Although not targeting all the features of commercial ISA, the toolchain is capable of compiling C code from LLVM intermediate representation or generating *m*OISC code translated from ARM, x86, RISC-V, and MIPS assembly. The toolchain also enables practical Value Change Dump (VCD) simulations output with graphical plots of the CPU state and associated symbols. A PoC microcontroller system has been synthesized in a low power Field Programmable Gate Array (FPGA) and verified in a basic wireless telemetry application including a Synchronous Peripheral Interface (SPI) RFM9x Long RAnge (LoRA) transceiver and a MAX30205 Inter Integrated Circuit (I^2C) temperature sensor, using its 8 bit I/O port, with software bus interface implementation. *m*OISC occupies ∼6% of resources on a Cyclone 10LP FPGA, for 1397 Adaptive Look-Up Tables (ALUTs) and 461 dedicated logic registers. The measured dynamic current consumption of the complete FPGA board with synthesized *m*OISC is 12 mA at 100 MHz clock.

INDEX TERMS One Instruction Set Computer, Microcontroller, Instruction Set Architecture, Compiler.

I. INTRODUCTION

ONE Instruction Set Computers represent the ultimate simplicity in the implementation of calculators [1], [2]. Notwithstanding that they run with only a single instruction, OISCs can implement Turing-complete machines, therefore, at the cost of higher execution time, they can solve any computing problem. A well-known Turing-complete OISC considers the `subleq` instruction [1]. In `subleq`, three memory cells named `a`, `b` and `c`, are accessed sequentially, to run both an arithmetic operation and control flow. `subleq` performs first an arithmetic operation, i.e., `mem[b] = mem[b]-mem[a]`, and based on the new value of `mem[b]` control flow is implemented conditionally: if `mem[b] <= 0`, then `pc = c`, otherwise `pc += 3`, where `pc` is program counter. Being an instruction that embeds both arithmetic operation and control flow, `subleq` can be classified as belonging to a CISC, although it is commonly referred to Reduced Instruction Set Computers (RISCs) [2]. This minimalistic approach to implement a computer (i.e., by compacting both data and control flow in a single instruction) has been even specialized using dedicated arithmetics to run encrypted and unencrypted computation [3]. However, OISCs are only mainly applied to computer teaching and in structural computing models [4], [5], thus, so far, used in domains at a significant distance from practical engineering.

`subleq` is not the only possible one-instruction capable of running universal computation. Besides Bit-Copying Machines [6], Transport Triggered Architectures (TTA) provide

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

another way to implement Turing-complete machines [7]. In TTA, the only instruction present is a `move`, and depending on the accessed memory address, the operation applied on their values changes to run, for instance, arithmetic operations or control flow. TTA architectures, with enough arithmetic and logic capability, notwithstanding their simplicity, have even been commercialized (refer to the MaxQ processor [8]). On the other hand, `subleq` schemes have not been implemented yet in practical computers, probably due to their reduced performance, hence, the necessity to run multiple instructions to implement ordinary operations required in practical programs. Indeed, in `subleq`, the emulation of a processor requires a large amount of memory, and typically multiple cores are required to provide acceptable performance [1]. Implementation of stack and function calls typically required in high-level languages are possible with `subleq` (see for instance the C Higher `subleq` compiler in [9], [10]). However, implementing a back-end for a modern compiler assuming a single instruction is not straightforward, even for low-complexity computers tailored to microcontroller applications. Moreover, a simple OISC, due to its aggressive minimalism, fails to consider power consumption aspects, that are instead fundamental in the design of current integrated systems.

The design of a tiny CPU core remains an interesting research topic, even today when microprocessors design is totally focused on performance and energy efficiency optimization, and designers just started considering, as a whole, production impact on sustainability [11]. First, Internet-of-Things (IoT) and Wireless Sensor Networks (WSN) applications require the revisitation of low-area-cost processors [12], [13]. Second, next-generation sustainable electronics, will probably require significant attention to silicon area [14]. Not so recent considerations on electronic systems fabrication, indeed, suggest that during a standard lifetime of a consumer electronic device, the energy burnt by the system is *lower* than the energy required by the fabs to produce its internal electronics [15]: consequently, assuming the same number of pieces, large silicon areas typically indicate larger environmental impact. Cross-sectional approaches such as Design Technology Co-Optimization have been considered to add sustainability in the power consumption, area and cost trade-off [16], thus enriching existing PACE analyzes [17]. These approaches consider multiple performance points to optimize a complete integrated system. However, from a practical viewpoint, it is evident that silicon area will remain a fundamental feature in next generation sustainable electronics, therefore posing the amount of combinational logic and registers required by digital circuits as a possible major player. Moreover, complexity plays a very important role as typically the environmental impact of semiconductor manufacturing is directly proportional to the number of metal layers used in the process back-end of line [18], hence on the interconnection complexity, by design. Thanks to its minimalism, OISC is an excellent paradigm for the implementation of von Neumann computers with non-traditional materials [19], and alternatives to ordinary silicon are highly demanded in sustainable manufacturing [20].

Modern microcontrollers are all dominated by reduced instruction sets, that enable efficient computation, pipelining, and low power consumption, by shifting complexity to the compilers. Notable examples that do not need introduction are ARM and RISC-V ISAs, that, although targeted for ultra-low power applications, are powerful enough, through their optimized microarchitectures, to run general-purpose operating systems. These ISAs and their associated hardware are supported by widespread consolidated software tools, compilers and optimizers that enable efficient code generation. Towards limited resources occupation, an open ISA that can be efficiently synthesized from Register Transfer Language (RTL) is the integer-only instruction set of RISC-V (RV32I) that has proven very efficient area and resource occupation on FPGA [21]. This instruction set can be widely extended adding ad-hoc instructions and customized even using commercial tool-chains engineered with leading compiler infrastructures. This notwithstanding, in general, the number of resources used to implement speed-focused RV32I microarchitectures is still considerable (see [22], [23]) assuming aggressive low-area requirements. With minimal synthesized features, for an RV32I the number of LUTs FF and memory elements required is on the order of 0.9 k, 0.4 k, and 1 k on an Intel Cyclone-IV FPGA [21]. Area optimized implementations are possible (see for instance PicoRISCV [24]), that are even synthesized with lower gate and register count. In light of this, *can a tiny CPU devised from a minimalistic OISC model, provide reduced resource occupation and run at a reasonable performance*?

The object of this work is to design a minimal but practically usable computer for basic microcontroller applications by extending OISCs. We present an ISA with an associated machine, named *multiple* One-Instruction Set Computer (*m*OISC, or *dynamic* RISC, *d*RISC, for similarity with ultimate RISC [2]) that can toggle among 14 run modes, each corresponding to a single OISC, with low resource occupation and based on a hybrid TTA/OISC scheme [25]. Notwithstanding its minimal instruction set, though with significantly lower performance compared to commercial processors, our computer can run basic microcontroller tasks to enable wireless telemetry applications and implement both SPI and I^2C interfacing based on C soft cores. Thanks to its simple organization, *m*OISC has the advantage of enabling straightforward compilation directly from LLVM Intermediate Representation (LLVM-IR), or direct code translation from other ISA. The known OISC in literature, are not capable of emulating commercial CPUs and running complex programs mainly because they require a large amount of memory in low resources FPGA [1]. By maintaining compatibility to `subleq` OISCs, *m*OISC, in a very limited logic resource budget, has the advantage of making it possible thus being capable of running code translated from x86, ARM, RISC-V, and MIPS assembly. It has the advantage of being modular and logic synthesis is possible including only the

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

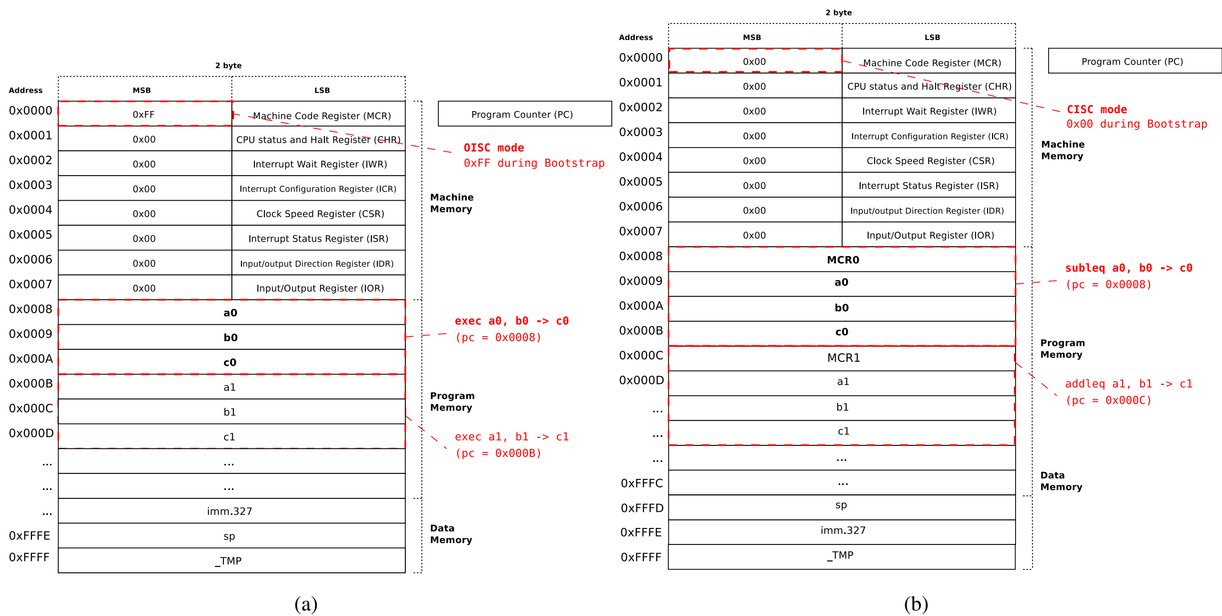


FIGURE 1. *m*OISC central memory `mem[·]` organization – OISC execution mode (a), and CISC execution mode (b).

instructions required by the application. *m*OISC implements a very simple but effective interrupt mechanism in which the CPU stops until a logic transition is detected at one or more I/O pins. Ultra-low power consumption with reduced switching activity is generally possible by waiting, for instance, that an energy conversion sub-system generates a stable voltage using a wake-up signal. We present the *m*OISC organization, a PoC microarchitecture, a basic compilation, code generation, and simulation toolchain, and we show results of a test case implemented using commercial I^{2}C and SPI chipset demonstrating a LoRA over-the-air transmission with an Intel Cyclone 10LP evaluation board and commercial rapid prototyping shields. Finally, we compare the resource occupation and the performance of the processor with respect to state-of-the-art open-source CPU implementations and simulation models.

II. *M*OISC ARCHITECTURE

*m*OISC is based on a single instruction that for simplicity, readability, and for its multiple significance we call *exec*. We assume that the computer utilizes only absolute addressing and a single memory that includes both instruction and data (von Neumann architecture). To pose a comparison with respect to well known architectures, *m*OISC provides an equivalent register-plus-memory addressing (with total orthogonality, [26]), i.e., the same instruction is valid for any memory address, registers included, without exceptions. The *m*OISC instruction format has the same notation of `subleq`-based OISC (i.e., `subleq a, b -> c`), and program memory can be compactly expressed as, `addr: exec a, b -> c`, where `addr` is the current memory address, `a`, is the source memory address, `b` is the destination memory address, and `c` is the jump memory address. `a`, `b` and `c` are also called *operands*. To run multiple OISC schemes, a specific register called Machine Code Register (MCR) is defined. *m*OISC can execute different run modes based on the value of MCR, hence, *exec* assumes different meanings, for instance, different OISC schemes such as `subleq` or `addleq`. This way, by tolerating the overhead of writing to MCR the machine issues at all effects instructions of multiple types but it can work as an OISC by statically setting a machine code without changes.

Based on the initial value of MCR during processor start-up, *m*OISC can support two execution modes. The first, called OISC mode, is compatible with state-of-the-art `subleq` machines (i.e., an instruction is encoded with 3 contiguous memory locations). The second, we call CISC execution mode, includes 4 subsequent addresses in program memory, with the first one storing the associated instruction MCR before the OISC memory addresses `a`, `b` and `c`. In this last run mode, the MCR register value is made explicit at each instruction. This trivial memory organization is inefficient from the code size viewpoint but has the advantage of being flexible, as the unused bits of the MCR can be used for multiple purposes, for instance, to implement other run modes or to further extend the number of machine codes.

Fig. 1(a) shows the organization of the memory of *m*OISC `mem[·]`, which is a Random Access Memory (and can be alternatively considered as a RAM-based register file) under an OISC execution mode. To enable non-volatile storage this memory can be implemented, e.g., as a Non-Volatile RAM (NVRAM). In our *m*OISC prototype, memory is 32768 elements wide (`0xFFFF`), and each cell has a width of 2 bytes for an overall of 64 kB. Each element value is represented in two's complement format. The *m*OISC, being a pure von Neumann architecture, has data and program memory

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

Address	Name	Register	Properties (RW/R)	Execution Effects
`0x00`	Machine Code Register	`MCR`	RW	R, W: non-blocking
`0x01`	CPU status and Halt Register	`CHR`	RW	R: non-blocking, W (`0xFF`): blocking
`0x02`	Interrupt Wait Register	`IWR`	RW	R: non-blocking, W: blocking
`0x03`	Interrupt Configuration Register	`ICR`	RW	R, W: non-blocking
`0x04`	Clock Speed Register	`CSR`	RW	R, W: non-blocking
`0x05`	Interrupt Status Register	`ISR`	R	R: non-blocking
`0x06`	I/O Direction Register	`IDR`	RW	R, W: non-blocking
`0x07`	Input-Output Register	`IOR`	RW	R, W: non-blocking

TABLE 1. *m*OISC memory mapped machine registers.

`b < 0x08`				
MCR	**Mnemonic**	**Name**	**Data Flow**	**Control Flow**
–	`MOV`	MOVe	`mem[b] = mem[a]`	`pc = c`
`b >= 0x08`				
MCR	**Mnemonic**	**Name**	**Data Flow**	**Control Flow**

`0xFF`	`SUBLEQ`	SUBtract and jump if Less or EQual	`mem[b] = mem[b] - mem[a]`	`if mem[b] <= 0:`
				`pc = c`
				`else:`
				`pc += 3 + u`

`0xEE`	`MOVLEQ`	MOVe and jump if Less or EQual	`mem[b] = mem[a]`	
`0xCC`	`ADDLEQ`	ADD and jump if Less or EQual	`mem[b] = mem[b] + mem[a]`	
`0x99`	`SHLLEQ`	SHift Left and jump if Less or EQual	`mem[b] = mem[a] « mem[b]`	
`0x88`	`SHRLEQ`	SHift Right and jump if Less or EQual	`mem[b] = mem[a] » mem[b]`	
`0x77`	`ORLEQ`	bitwise OR and jump if Less or EQual	`mem[b] = mem[b] | mem[a]`	
`0x66`	`ANDLEQ`	bitwise AND and jump if Less or EQual	`mem[b] = mem[b] & mem[a]`	
`0x55`	`XORLEQ`	bitwise XOR and jump if Less or EQual	`mem[b] = mem[a] ^ mem[b]`	
`0x44`	`XNORLEQ`	bitwise XNOR and jump if Less or EQual	`mem[b] = ~(mem[a] ^ mem[b])`	
`0x33`	`PC`	Program Counter save	`mem[b] = pc`	
`0x22`	`MEM`	MEMory double-depth addressing	`mem[mem[b]] = mem[a]`	`pc += 3 + u`
`0x11`	`MEMR`	MEMory Reverse double-depth addressing	`mem[a] = mem[mem[b]]`	`pc += 3 + u`

`0x00`	`PCS`	Program Counter Set	–	`if mem[b] == 0:`
				`pc = c`
				`else:`
				`pc = mem[b]`

TABLE 2. *m*OISC machine run modes (ISA) for the execution of the generic instruction `exec a, b -> c`, that are set by writing to MCR.

collapsed in the same storage unit. Memory is divided into three logic parts: *Machine Memory* (or, registers), *Program Memory*, and *Data Memory*. While the 8 addresses `0x00`–`0x07` are fixed, the program and data memory size depend on the program to be run. Each machine register (where the first is the above-defined MCR) has a specific function (see Sec. II-A and II-B for further details).

Fig. 1(b) shows the organization of the internal memory in the *m*OISC processor `mem[·]` under a CISC execution mode. The organization is the same as for the OISC execution mode, with the only difference in the opcode and operands packing. In CISC mode the MCR is made explicit and consequently, the program counter is advanced by 4 and not 3 as in OISC mode.

The machine registers depicted in Fig. 1(a) and Fig. 1(b) are summarized in Tab. 1, with read/write or read-only properties. Besides run mode, machine registers are used to set and read the I/O port, its input-output direction, the internal clock speed of the machine, to stop the CPU while waiting for an I/O event on physical pins, and to read the last arithmetic comparison results between `mem[a]` and `mem[b]`, including overflow status.

A. MACHINE CODE REGISTER AND *m*OISC RUN MODES

Tab. 2 shows all the possible machine run modes supported by *m*OISC. At program counter level, the difference between OISC and CISC execution modes is only given by the amount of its increment. We use an additional number `u` to identify the run mode of the machine: if `u` is 0, the program counter is advanced by 3 memory addresses (OISC mode), and if `u` is 1 program counter is advanced by 4 addresses (CISC mode). Observe that the execution of the machine does not change between these two modes, as it differs only in the program counter increment.

We have chosen to design *m*OISC with 14 run modes, a good trade-off between simplicity and the performance level required to manipulate integer data and implement a basic bus interfacing. In OISC mode, after the MCR is set with a specified 1 byte constant, and it is not overwritten, all subsequent instructions will be of that type and the run mode is maintained. We here report all the run modes. In the description of each one we refer to a generic program line identified as `exec a, b -> c`, with the operands `a`, `b`, `c` defined previously. The processor program counter is identified as `pc`. Since in *m*OISC each machine register has a width of 1 byte, when MCR is set, e.g., an instruction

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

exec const.14, MCR is issued (where const.14 is a constant value in the data memory), the processor considers the lowest 8 significant bits of the source operand, in this case implicitly computing 0x00FF & const.14. When operand b is within 0x00 and 0x07, the machine always operates as a move machine irrespective of the current setting of the operational machine mode, but without control flow (i.e., always pc = c). In this run mode and for these memory regions, the computer can be defined as transport triggered. With respect to subleq which implements universal computation, we have chosen to include other elementary instructions to speed up computation in practical applications and save memory.

Arithmetic and logic operations, i.e., SUBLEQ, MOVLEQ, ADDLEQ, SHRLEQ, SHLLEQ, ORLEQ, ANDLEQ, XORLEQ, XNORLEQ, apply the same conditional control flow of state-of-the-art subleq OISC, by simply assuming a generic operand *op*, for an *op*LEQ data flow in the form mem[b] *op* mem[a]. This control flow is applied also to the instruction PC. All the aforementioned arithmetic and logic machine codes are not strictly necessary to enable practical programming: only SUBLEQ, MOVLEQ, PC, MEM, MEMR, and PCS are enough to implement computing, notwithstanding lower performance compared to machines with a larger number of hardware-accelerated instructions.

Apart from the OISC instructions SUBLEQ and ADDLEQ and the remainder logic instructions in which the same control flow of subleq is applied, we provided the last four instructions to easily implement stack and function calls. The instruction PC (with no conditional control flow on the value of mem[b]) saves the current program counter on a generic memory cell mem[b], where b can be any memory address except machine memory (i.e., b > 0x07). a is a dummy operand and is not considered during execution. This instruction can be useful to store the program counter value before jumping to a determined function. Practically, the stored value needs to consider the memory offset required to execute the jump therefore, in general, it needs to be updated based on the instructions between the PC instruction address and the jump instruction address to a function. Observe that PC is a simple move instruction where mem[a] is program counter.

The instruction MEM saves the content of a memory cell in another memory cell whose address is specified, in turn, in a third memory cell x, i.e., x = mem[b], mem[x] = mem[a] where a and b can be any memory address. As the program counter is automatically advanced, c is here a dummy operand. This instruction is useful to emulate the stack in the main *m*OISC memory. The stack can be very simply represented by a data memory cell, for instance, called m_ptr, that is initialized with a value corresponding to the stack pointer at the beginning of the program. By adding an offset to m_ptr and by applying the MEM instruction on it, it is possible to read data from a generic memory cell. We name this instruction as double depth addressing because it operates with two nested memory addressing. This instruction can be applied to any memory cell, including machine registers, except from MCR (i.e., mem[b] > 0) that cannot be updated using MEM. Since MEM is typically used to access data values, we have chosen to exclude MCR to reduce the number of comparisons applied on the value of mem[b], thus conceptually simplifying our control unit.

The instruction MEMR saves the content of the memory whose address is specified in another memory cell x, i.e., x = mem[b], mem[a] = mem[x] where a and b can by any memory address. Similarly to MEM, the instruction cannot be applied to MCR (i.e., a > 0). MEMR is the complementary instruction of MEM, and it can be used to pop values from the stack by using a pointer variable and subsequently decreasing it by a specific offset. Observe that MEM and MEMR, given the elementary architecture of this computer, work at all effects as a load and store operation in typical RISC and CISC architectures. In conventional processors typically a limited-sized register file is mostly accessed, while data from memory, that requires a larger number of clock cycles to be retrieved or saved, is specifically accessed through dedicated load and store instructions. In *m*OISC, given its absolute addressing, the memory can be considered, indeed, as a large RAM-based register file. Finally, the instruction PCS sets the program counter with the value specified in mem[b]. This instruction can be effectively used to return from function calls. The program counter is set only if mem[b] is non-zero.

B. OTHER MACHINE REGISTERS

The CPU status and Halt Register (CHR, address 0x01) provides the overflow status for the last executed SUBLEQ or ADDLEQ operation, provides the operands comparison flags, and halts the machine indefinitely until a new hardware reset is issued. The machine is stopped only when 0xFF is written to CHR. An overflow condition is detected only when a SUBLEQ or ADDLEQ operation is run. If overflow occurs, the CHR least significant bit (overflow flag, 0x01) stays '1' until the next correct SUBLEQ or ADDLEQ is executed. Every time an arithmetic SUBLEQ or ADDLEQ operation is run the 4th, 3th, and 2th least significant bits (operand flags) are updated based on the last processed values mem[b] and mem[a]. If mem[b] > mem[a] then the value 0x02 is set (b is bigger than a). If mem[b] < mem[a] then value 0x04 is set (b is smaller than a). If mem[b] = mem[a] then value 0x08 is set (b is equal than a). Multiple bits set can occur depending on the values of mem[a] and mem[b]. In any other condition, CHR remains latched to the previous value. CHR can be always read without impacting the CPU control flow.

Writing IWR (address 0x02) stops the processor and waits for an interrupt from a physical pin whose transition direction 0-to-1 or 1-to-0 is specified in the Interrupt Configuration Register. The IWR bits are associated directly to a physical IOR I/O pin, from 0 to 7. The bits set to 1 enable the CPU to filter the expected pins for the specific transitions specified using ICR. For instance, if IWR is set to 0x05,

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

the CPU will resume only if a logic transition occurs at `IOR2` or `IOR0` (because `0x05` is `0b00000101`). The IWR setting is valid only if the direction of the IOR pins is in read-mode. Particular attention must be taken while writing IWR in moments other than bootstrap. For instance, if the value `0x00` is written the effect is the same as a CPU halt without recovery from any transition (at least one bit needs to be set to make the machine sensitive to a logic transition). Moreover, in case of interrupt wait on an I/O pin that has been configured as an output, the requested condition will never be reached, still resulting in a CPU halt.

ICR (address `0x03`) sets the transition direction 0 to 1 or 1 to 0 for all the IOR pins of the CPU to be detected during an interrupt wait trigger (see IWR). Bits set to logical 0, identify a 0-to-1 transition while bits set to logical 1, identify a 1-to-0 transition. For instance, if ICR is set to `0x07`, logic transitions expected for `IOR2`, `IOR1` and `IOR0` are 1-to-0 while for `IOR3`–`IOR7` are 0-to-1 (because `0x07` is `0b00000111`). The ICR setting is valid only if the direction of the IOR pins is in input mode.

CSR (address `0x04`) sets the current CPU clock frequency. The CPU speed is set on the fly, that is the clock is immediately toggled without any delay given by the hardware implementation of an internal Phase Lock Loop (PLL) or, alternatively, an oscillator array. The possible CPU speeds are hardware and implementation-dependent. ISR (address `0x05`) contains the indication of the pin that is set after an IWR write occurred. The register is set immediately after the specified trigger occurred and the value is made available for the instruction immediately succeeding the previous IWR write. The bit set to logical 1, identifies that the processor has been woken up on the specified IOR pin (that implicitly toggled with the transition direction specified in ICR). The IDR register (address `0x06`) sets the input/output direction of all IOR pins. Bits at logical 0, are in input mode, bits at logical 1 are in output mode. IDR bits are associated directly with a physical IOR I/O pin.

The IOR register (address `0x07`) sets and reads the current output and input logic values of the I/O pins. IOR bits are associated directly to a physical IOR I/O pin, from 0 to 7. Observe that an IOR write is masked using the IDR register, therefore if IDR = `0xF0`, setting the IOR register to `0x0F` has the effect of zeroing the high nibble because the 4 least significant bits are in input mode (IDR = `0b11110000`). A read-write operation on IOR is always non-blocking.

C. ASSEMBLY LANGUAGE

We have considered a sufficiently simple assembly language to to be able of writing basic programs without the need of high-level languages and associated toolchains. We have maintained a case sensitive syntax for compatibility with Linux/Unix systems conventions. To increase code readability, comments can be entered using the character `#` which is used to turn a complete line into a comment. In the assembly file characters `\n`,`\r` are allowed and do not represent addresses (the same as for `#`, that is, an empty line in the assembly does not identify a data memory address or an instruction). To make the code more readable, the programmer can use `\t` or spaces to separate labels with instructions/data. In general, every assembly line identifies an address. Addresses are encoded incrementally where address `0x0000` is the first non-commented line. The first 8 addresses (`0x0000`–`0x0007`) must be reserved for machine registers that are placed at the beginning of the program and must be specified all, with no exceptions. The value of `IWR` is discarded during the bootstrap sequence, i.e., when program execution starts for program counter equal to zero. Starting from address `0x0008` program memory can be specified. `addr` can be also omitted as it can be statically computed at assembly time. If operand `c` is unspecified, it is implicit that `c` contains the next instruction address (that is added by the assembler program). Operands `a` and `b` are mandatory.

When the program memory is concluded, data memory can be appended next. There are no limitations for both program and data memory size as long as they fit `mem[·]`. In *m*OISC a single data memory cell `mem[x]` stores one operand. The data memory is then a collection of variables with symbolics that can be utilized as operands `a`, `b`, `c` in the program memory section. The *m*OISC data memory can be expressed using the notation symbolic-address**:**value to identify the content of the memory cell. Each data memory line is a program variable (indeed called symbolic address), whose address is calculated at assembly time. The data memory needs to be declared after the program memory, with the only exception of the first 8 addresses from `0x00` to `0x07` that include the initial value of the machine registers. Variables, i.e., symbolic addresses, are alphanumeric but they cannot start with a number. Valid names are, e.g., `t0`, `line1`, `lab.1`, `_a`, `bB`, `$g.56`. The character `-` is used to identify negative numbers and cannot be used for variable names.

All values in the machine, program, and data memory sections must be specified as decimal signed numbers.

1) OISC Execution Mode

In OISC mode, the only instruction that is allowed is `exec`. An example of *m*OISC assembly is given in Lst. 1.

```
1  # Machine registers section
2  MCR:    255
3  CHR:    0
4  IWR:    0
5  ICR:    0
6  CSR:    192
7  ISR:    0
8  IDR:    0
9  IOR:    0
10 # Program memory section
11         exec _SUBLMCR, MCR
12         exec _NULL, _NULL -> main
13 memcpy: exec _SUBLMCR, MCR
14         exec _NULL, _NULL
15         exec _MEMRMCR, MCR
16         exec Var_2_memcpy, m_ptr
17         exec _SUBLMCR, MCR
18         exec const.0, m_pt
19            ...
20         exec _MEMMCR, MCR
```


This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access


```
21         exec const.20, Var_33_main
22         exec _SUBLMCR, MCR
23         exec _NULL, _NULL -> Label_19_main
24 # Data memory section
25 _SUBLMCR:         255
26 _MEMMCR:          34
27 _MEMRMCR:         17
28 _NULL:            0
29 m_ptr:            31500
30 _NULL:            0
31 link_register:    0
32 _TMP:             0
33 const.0:          1
34 Var_2_memcpy:     0
35 Var_33_main:      0
36 const.1:          2
37 const.20:         -3
38 ...
```

Listing 1. Example snippet of *m*OISC assembly code (OISC mode).

The first 8 assembly lines specify the values of the machine registers at start-up time. In this example `MCR` is 255 to set the machine execution mode as OISC. The first instruction of the routine `memcpy` is specified as a symbolic address so that the subsequent code can implement jumps to it. All the data memory identifies both variables and constants with a specific symbolic address. Observe that it is possible to have a mix of program and data memory in the assembly (therefore, without keeping these two parts strictly separated). This can be done only if the preceding and subsequent `exec` instructions above and below a given data memory region implement unconditional jumps to wrap data around. The size of a data memory variable is 2 bytes (16 bit), while the size of an instruction is $3 \cdot 2$ bytes (48 bit).

2) CISC Execution Mode

In CISC mode, the internal organization of the program memory is modified, while data memory is unmodified. Indeed, instructions are packed using 8 bytes, in particular given an instruction n^{th}, for each instruction the information `MCR`, `a`, `b` and `c` is stored. The assembly syntax and conventions remain the same of the OISC mode, with the difference of explicitly defining the setting of the MCR. For instance, the following OISC mode code (machine registers not shown),

```
1   lineE: exec mcr0, MCR
2   lineF: exec a0, c0
3   lineG: exec mcr1, MCR
4   lineH: exec a1, c2
5   lineI: exec a3, d4
6   lineJ: ...
7   ...
8   mcr0:  255
9   mcr1:  238
```

is equivalent to the following CISC mode code,

```
1   lineF: subleq a0, c0
2   lineH: movleq a1, c2
3   lineI: movleq a3, d4
4   lineJ: ...
5   ...
```

where `subleq` and `movleq` are the operational codes (assembly mnemonics) associated to the values 255 and 238 of MCR (i.e., `0xFF` and `0xEE`), respectively. When operating for $\mathrm{b} < 8$, i.e., assuming a `mov` instruction, for simplicity a `movleq` identifier is used. OISC mode assembly files can be automatically converted in CISC assembly files by storing the last value of MCR (that is set by `exec <addr>, MCR`, where `<addr>` is a memory address that stores a valid machine code) and applying it to the next instructions in sequence. This type of translation is always applicable with the foresight of making explicit the MCR at every possible label. This way an OISC code can be translated into CISC mode using a simple sequential scan.

III. PROOF-OF-CONCEPT IMPLEMENTATION

A. BLOCK SCHEME

Fig. 2 shows a block scheme of a proof-of-concept implementation of the *m*OISC ISA, including the 1 byte input/output port, internal PLL for clock generation, and reset pin. The design has been synthesized in a low-cost Cyclone 10LP device with sufficient M9K block-RAM elements to support 32768 addresses at 16 bit data for central memory [27]. The PLL, I/O Buffer, and Memory block are built-in mega-functions provided by the Quartus™ design software. The proof-of-concept hardware does not include any bus and it has been kept trivial to enable the easy verification with the compiler and simulator that will be presented later on. All the code has been written in RTL form. The processor firmware is flashed by programming the Cyclone 10LP device through initialization of all the RAM elements through a `.mif` file generated by the compilation toolchain.

The microarchitecture accepts the main 50 MHz clock provided by the Cyclone 10LP evaluation kit to generate four different clocks, 100 MHz, 50 MHz (re-clocked), 1 MHz, and 10 kHz, for respective CSR values `0x00`, `0x40`, `0x80`, and `0xC0`. The clocks feed a multiplexer controlled by both CSR and a halt signal that completely gates the clock propagation when the machine is halted. The multiplexed system clock `CLK` is propagated in all internal sub-systems, comprising a system `RST` signal that is derived from the external reset pin of the FPGA through a debouncer (synthesized assuming trigger event filtering of 16 `CLK` cycles). The debouncer filters also the `HALT` signal from the CPU. The Arithmetic Logic Unit (`ALU`) accepts as inputs the current `MCR` value and the memory elements determined during instruction fetch, i.e., `mem[a]` (MEM_A), `mem[b]` (MEM_B) and `mem[mem[b]]` (MEM_MEM_B), used in double depth addressing. To execute the `PC` instruction the `ALU` accepts also the program counter `PC` as input so that it can generate the value of `DATA` to be stored in memory during the CPU cycle. To identify overflow and provide comparison flags, the `ALU` provides also `OVERFLOW` and `CMP`, 1 bit and 3 bit respectively (greater, lower or equal). To permit set-up and hold timing constraints closure, all the three outputs of the `ALU` are sampled used dedicated registers.

In this implementation, the single *m*OISC memory is a single port RAM with a parallel interface and includes write and read enable signals `WREN` and `RDEN`. This interfacing normally matches commercially available NVRAM that can

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

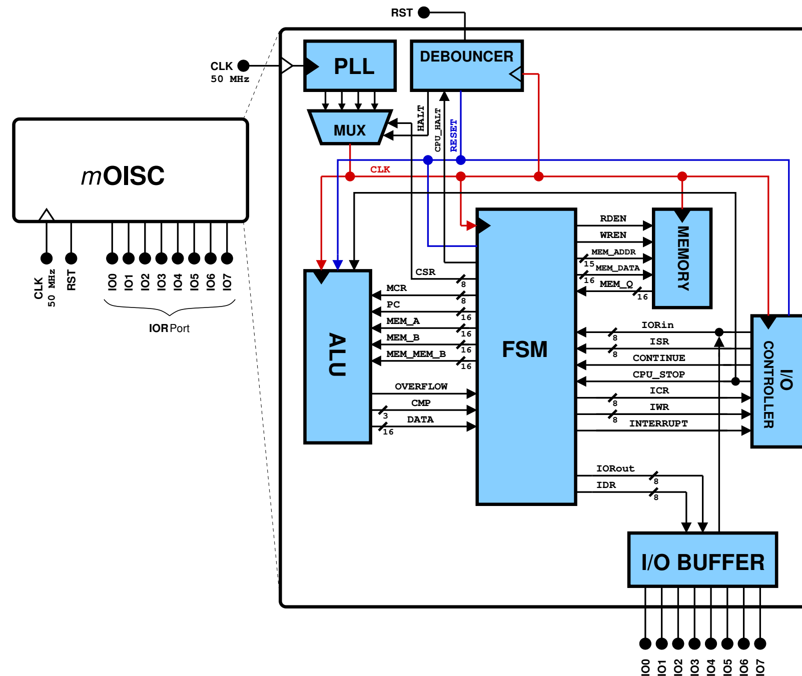


FIGURE 2. *m*OISC proof-of-concept microarchitecture block scheme.

be utilized in future prototypes of the processor, for instance with optimized microarchitecture or area occupation. Given the low availability of internal RAM on the used Cyclone 10LP FPGA, the memory is addressed by 15 bit only (MEM_ADDR), while data width is 16 bit (MEM_DATA and MEM_Q). The interrupt mechanism of the CPU is handled by the I/O CONTROLLER that utilizes a number of register values to implement a wait cycle on the input port, to filter the required transitions (low-to-high or high-to-low) and block the CPU until these transitions are detected. The I/O CONTROLLER considers the input data from the I/O BUFFER (that provides separate input and output ports for driving and reading physical pins defined by the IDR register), ICR, and IWR to mask the I/O pins and set the transitions edge type. The controller is invoked by using a dedicated INTERRUPT signal that stops the CPU by rising a CPU_STOP signal and by continuously determining the transitions at the I/O BUFFER until one of them satisfies the trigger condition. When detected, the controller sets the ISR accordingly to the I/O pin that actually triggered the interrupt (useful for the SW in case more than one I/O pin has been activated in the IWR), returns in a wait state for a successive interrupt trigger and issues signal CONTINUE for the CPU to continue execution.

B. FINITE STATE MACHINE

The control unit of the CPU is implemented using a single Finite State Machine (FSM) that uses the input and output signals provided by the ALU, the I/O CONTROLLER, and I/O BUFFER, to implement fetch, decode, execute and write back on the single-port RAM. Although an FSM is definitely not an optimized solution to implement a CPU control unit, it permits ease of debugging and verification, and moreover permits to flexibly add additional instructions or remove the unnecessary ones as will be shown later.

Fig. 3 shows a simplified scheme of the *m*OISC internal FSM. Execution mode is stored in an internal variable u that can have the value 0 or 1 in OISC or CISC mode, respectively. The CPU cycle starts assuming u is zero. First, *m*OISC reads the value of the IOR provided by the I/O port and stores it in the corresponding internal register. Machine registers are named MR[] in this diagram (for instance MR[0] is MCR, MR[1] is CHR, MR[2] is CHR, and so on). To compactly express the use of IDR, i.e. MR[6], every time IOR is written, a bitwise AND operation is assumed to be executed (the IDR bits at logic '1' identify an output direction). After updating IOR, fetch is implemented in 6 clock cycles, that is, reading a, b, c, mem[a], mem[b] and mem[mem[b]]. After fetch, the FSM jumps to different states according to the value of the program counter. If pc<8, the bootstrap sequence is executed, and the value of mem[a] is first stored in internal registers MR. If the program counter is zero (i.e., the current pc is pointing to the initialization of the MCR), the value of u is updated to save machine execution mode and the program counter is increased by 1 to scan all the machine register region. The bootstrap sequence continues until pc reaches the value of 8, i.e. when the machine can execute instructions normally. In CISC mode, during normal execution and even during bootstrap whereafter u has been set, the fetch phase considers the read and the storage of the MCR in the internal register MR[0], preceding the reading of a.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

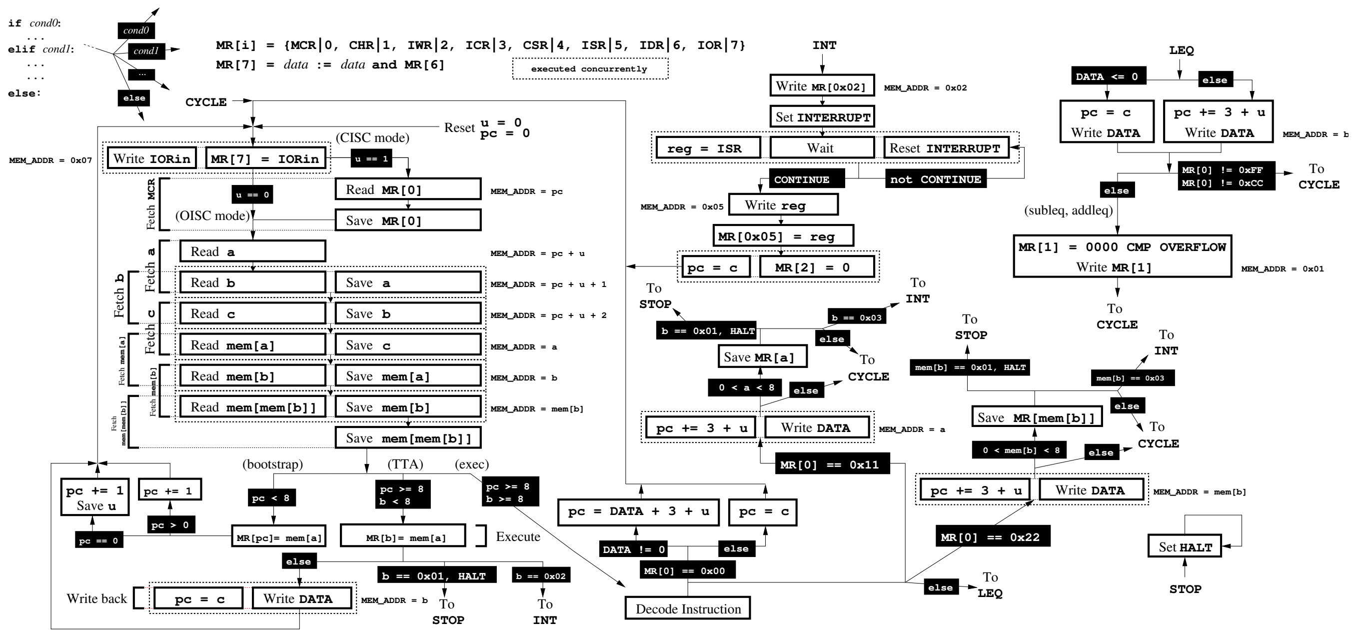


FIGURE 3. *m*OISC proof-of-concept microarchitecture FSM with detailed state transitions.

For $\mathtt{pc} \geq 8$, two possible executions are possible, one for b<8, and another otherwise. $\mathtt{b} < 8$ indicates that the current instruction will write its output to a machine register, in the diagram indicated as TTA. The corresponding internal register MR[b] is then updated with the value mem[a]. At this point, the CPU needs to handle three special cases, that are i) a CPU stop, issued when b is 1 and mem[a] is 255, ii) wait for interrupt instruction occurring for $\mathtt{b} = 2$, and otherwise iii) write back mem[b] in the machine register. In the first case, the FSM jumps to a state where the STOP is issued (CPU_HALT in Fig. 2). In the second case, (see INT arrow in the diagram), the CPU writes back in the memory the value of MR[2] (using a memory address 0x02), sets the INTERRUPT signal, and waits indefinitely until the CONTINUE signal is asserted by the I/O CONTROLLER. At the same time, it stores the current ISR value provided by the controller in a temporary register reg. When CONTINUE is asserted the content of reg is written back in memory at the address 0x05, and in the internal register of the CPU MR[5]. Next, the value of the IWR (i.e, MR[2]) is reset to zero, the program counter is set to c, and the CPU is ready to start a new cycle. In the third case, the machine simply needs to implement write back to the memory and need to set the program counter to the address c before restarting a new CPU cycle.

When $\mathtt{b} \geq 8$, the machine needs to implement the normal flow to handle operation (exec in the figure) and as the first step, it decodes the instruction. The FSM next state depends on the instruction type, and here four cases are possible, i.e., PCS, MEMR, MEM, and all the remainder instructions. In the case of PCS, that is $\mathtt{MR[0]} = 0$, the program counter is updated with the output DATA of the ALU, which is mem[b]. Here two cases need to be handled according to the value of DATA. The machine needs to check if DATA is not zero and the PCS instruction is implemented only if DATA is not zero, otherwise, the operation is skipped and the program counter is updated with the value of c. In the case of MEMR and MEM, that is $\mathtt{MR[0]} = \mathtt{0x11}$ and 0x22, the CPU needs to execute the instruction by assuming that both a and mem[b] can fall within the range of machine registers, and re-implement the TTA control. Indeed, after write-back to memory (at address a or mem[b], respectively) and program counter increment, the system needs to consider again a and mem[b] in case their value is lower than 8 and larger than 0 (i.e., MCR). In such a case, the same condition of TTA applies again, i.e., for CPU halt and wait for interrupt trigger. In case the current instruction is other than the above, the LEQ states are executed. Observe that the control flow associated with the arithmetic and logic instructions is also valid for the instruction PC because the program counter is simply made available on DATA by the ALU. After program counter update and write back, that vary based on the value of DATA, in case of a SUBLEQ or ADDLEQ instruction is executed, the machine needs to update the flags in CHR (i.e., MR[1]). CMP and OVERFLOW, which are provided by the ALU, are used to update MR[1] and to execute a write back on memory. After these last operations, the CPU can execute another cycle.

IV. COMPILER AND SIMULATOR TOOLCHAIN

We have developed a complete simulation, RTL generation and compilation toolchain with custom stages in Python to increase portability across multiple operating systems. In particular, we developed i) a bytecode simulator outputting a VCD file that can be easily read using open-source viewers

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

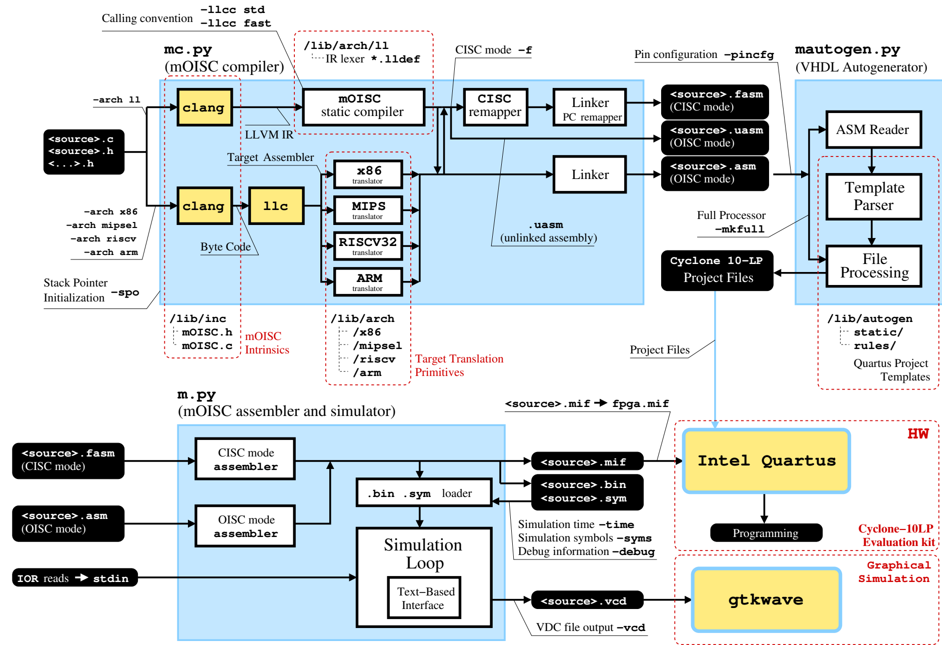


FIGURE 4. *m*OISC compilation and simulation toolchain implemented using Python 3.

such as GTK Wave, ii) a C compiler/translator that processes LLVM IR from `clang` to automatically generate assembly code or, alternatively, translates assembly from LLVM targets (`x86`, `mipsel`, `riscv32`, and `arm`) to *m*OISC, iii) an assembler program that generates binary files for the simulator and a MIF file to be passed to the FPGA synthesizer for hardware validation. The toolchain requires at least LLVM 9.0 and GTK Wave to operate, which are available both under Linux, MS Windows, or macOS. The compiler/translator aims at supporting integers, integer vectors and pointers, enough for a simple data transfer application. It does not implement all opcodes from the targets nor all IR instructions. Adding further supported types such as `char` or `struct` declarations is possible and it does not require extensions of instruction set of the machine. In this work, we have used LLVM to compile source code and generate both intermediate representation and commercial architecture assembly [28]. In case of LLVM-IR compilation, only the LLVM `clang` front-end is used. Compared to other works where processors have been directly ported within an internal LLVM target (see, e.g., [29]), here, we have re-used the existing LLVM targets.

Fig. 4 shows a conceptual block scheme of the complete *m*OISC compilation and simulation toolchain. For the sake of brevity, we herein report a high-level description of the tools without going into specific details.

A. *M*OISC COMPILER

Targeting low-complexity applications, we consider the compilation of a single source file (`<source>`), in this example called `sensor.c`, the associated header file, and a set of include files that define machine intrinsics in a `/lib/inc` subdirectory. The include and associated file `mOISC.h` and `mOISC.c` define the address of machine registers and the built-in function `memcpy`. The *m*OISC machine registers are defined as volatile integer pointers. This way, reading and writing such registers can be achieved using simple assignment statements, without the need of any ad-hoc built-in instruction, as normally done in commercial microcontrollers. The LLVM built-in function `memcpy`, normally referenced for x86 architectures assembly, is explicitly defined to simplify assembly translation. The C-to-assembly compilation is achieved using `mc.py` that internally defines two flows, one that implements LLVM-IR compilation and another that uses LLVM assembler output to translate it into an *m*OISC assembly. Both flows consider the `clang` front-end to generate IR or bytecode, respectively, and an optional `-spo` argument (otherwise at default value `31500`) initializes the value of the stack pointer implemented as a simple variable in the data memory. All the C code is compiled by `clang` and `llc` (for direct translation) without optimization (option `-O0`). For simplicity, we assume that all variables are declared as `int`. *m*OISC operates at 16 bit and for both compilation and translation we simply represent data on 16 bit only. For a basic compilation functionality, LLVM intermediate representation is abstract enough to allow the non-consideration of types in the translation: for MSP430 targets (intrinsically at 16 bit) IR arguments are automatically outputted with `i16` and `*i16` types, while for ARM for example, arguments are 32 bit. Besides types, the intermediate representation does not significantly change without optimization flags, hence relaxing complexity for *m*OISC code generation. In direct translation, for instance, in x86 architectures, we considered only the least significant word of 32 bit registers such as

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

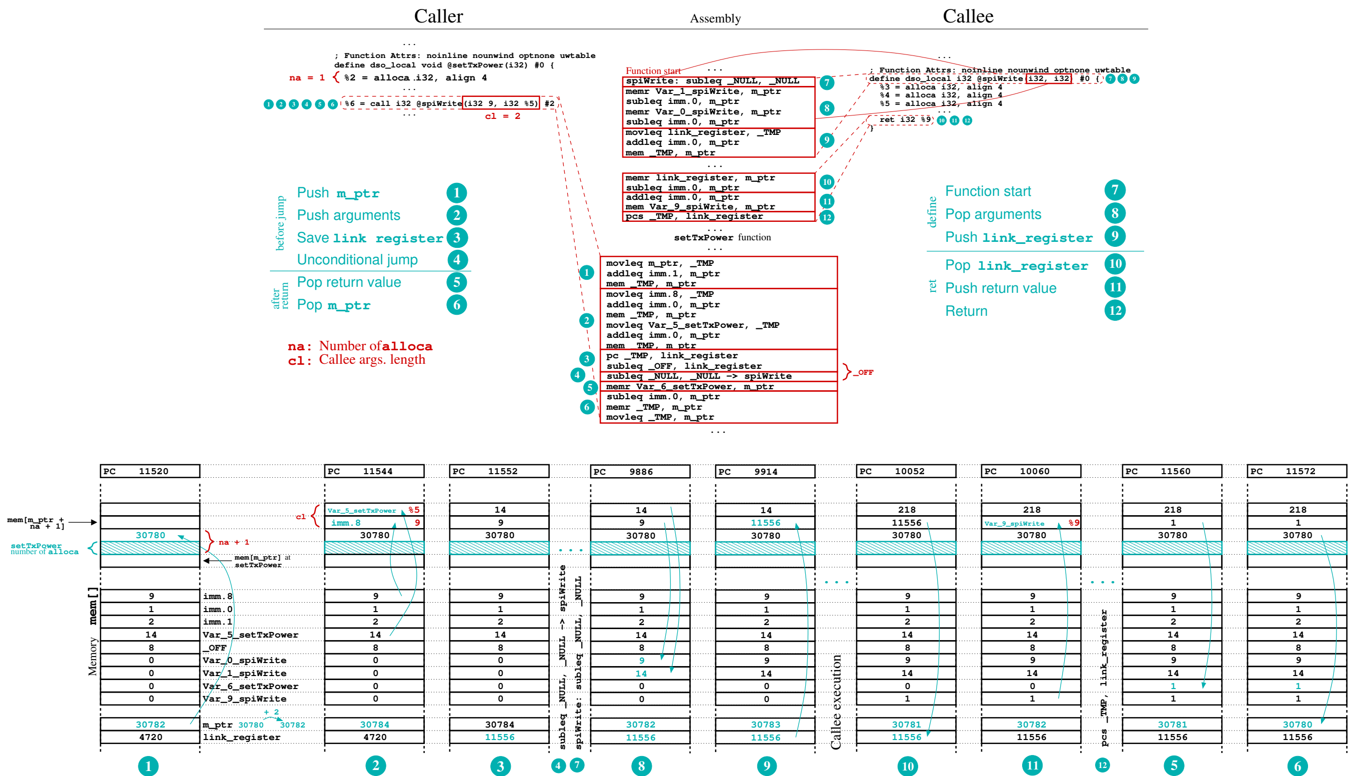


FIGURE 5. *m*OISC compiler standard calling convention implementation based on an example `mc.py` compilation output in CISC mode.

EAX, EBX, ECX, and EDX, and we truncate negative signed numbers accordingly.

Observe that thanks to the ultra-simplified addressing mode of the machine and the absence of register file, the compilation from LLVM-IR and the translation from commercial CPU assembly is straightforward: each register in the Static Single Assignment (SSA) graph, can be considered at all effects as a data memory variable in *m*OISC. LLVM-IR, indeed, is based on a register machine with an infinite number of registers [30], that perfectly matches the memory organization of our computer. A flat operand translation both from assembly and LLVM-IR instructions is thus possible without having to track and re-map the effective number of registers in use in a register file, as needed for instance for other ISAs.

1) Basic LLVM-IR Compiler

In the case of LLVM-IR compilation (argument `-arch ll` passed to `mc.py`) the software reads a set of lexer primitives that define the syntax of the commands in a subfolder `/lib/arch/ll`. Internally, it implements two different calling conventions, standard and fast (argument `-llcc std` or `-llcc fast`, respectively). The standard calling convention, which uses the stack to pass function arguments from the caller to the callee and to save the return value, is detailed next. The fast calling convention, similar to the standard, not using the stack but simple move instructions to copy caller variables to callee inner variables, is not reported for the sake of brevity. To implement a basic compilation for simple programs we have considered the LLVM-IR instructions `add`, `define`, `or`, `alloca`, `getelementptr inbounds`, `and`, `global`, `ret`, `ashr`, `icmp`, `sext_to`, `bitcast_to`, `shl`, `br`, `inttoptr`, `load`, `store`, `call`, `sub`, `constant`, labels and `memset` intrinsic.

The LLVM-IR compiler outputs, for debug purposes, an unlinked OISC mode code with extension `.uasm` in which certain references to data memory cells (for instance LLVM `global` definitions) are not yet substituted with absolute memory values. The final OISC code generation is completed by the `linker` internal module, that calculates absolute addresses and substitutes symbolic values with the correct memory address to generate `sensor.asm`, the OISC mode assembly that can be processed by the next module. In CISC mode, the unlinked `.uasm` assembly is post-processed by linking symbolics with a different convention compared to OISC mode, because memory cells have all different addresses compared to the OISC mode. In CISC mode, moreover, the `linker` performs also a program counter remapping by parsing and updating a reserved `_PCMCR_RETADDRBL` variable used in the execution of call returns. This has two different values in OISC mode and CISC mode because instruction length is different. CISC mode compilation, which generates a corresponding `.fasm` file, is invoked using a `-f` flag at the command line.

Fig. 5 shows our implementation of the standard calling

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

convention from the LLVM-IR code generated by `clang`. The stack pointer (herein defined as memory pointer) is identified by the variable `m_ptr` that saves the address of the top of the stack. Let us assume that from a function called `setTxPower` that accepts one `i32` argument, another function called `spiWrite` having two arguments is called. The compiler needs to keep track of the current number of `alloca`, compactly `na`, that is the number of stack variables of the current function, in this case, `na`= 1. To implement the function call the compiler needs to push the current stack pointer not to overwrite the stack data of the caller, at an offset exceeding 1 of the caller `na`. In our simple implementation, we assume that the number of `alloca` can be statically computed at compile time. This is reasonable for the present application domain and simple micro-control code. With this hypothesis, after lexer and parser execution we scan the complete file and populate an array of structures, one for each function, in which both `na` and the number of function arguments are stored. In our implementation, the `alloca` variables address is computed by adding an offset to `m_ptr`, thus not defining separate variables for them. In general, `na` needs to be computed by the program at runtime because a function may directly allocate memory on the stack. Under this hypothesis, the calling convention described further on still operates but an additional variable `na` needs to be defined by the compiler, per function, to update the number of allocated bytes in the stack.

When a `call` instruction is generated by the compiler, i) the value of `m_ptr` is increased by `na`+ 1 and the current `m_ptr` value is pushed (in this example, using `imm.1`= 2), and ii) the compiler pushes arguments in the stack in reverse sequence, for an overall number of pushes corresponding to the number of callee arguments (`c1` in this example). In this example, two arguments need to be pushed, a constant value `9` (`imm.8` in this example) and an inner caller variable `%5`, that is identified by the compiler with suffix `Var_` and postfix `_setTxPower`, i.e., `Var_5_setTxPower`. Next, step iii), the compiler saves the new `link_register` that stores the return address from the callee. The `link_register` stores the current program counter `PC` with an offset `_OFF` (that is `_PCMCR_RETADDRBL`) that takes into account the length of the next instructions to implement unconditional jump. In this example `_OFF` is subtracted from `link_register`, therefore it is a negative number. The unconditional jump, step iv) of the caller, is simply implemented using a `_NULL` variable, subtracted by itself through `SUBLEQ` that points to the label of the callee, `spiWrite`.

At this point, the stack includes both arguments of the callee, the `m_ptr` of the caller, and the variable `link_register` includes the return program counter value after the callee returns. Let us consider the callee side, starting from step vii). Although wasting one CPU cycle, for simplicity the function start is generated by issuing a label and an equivalent `SUBLEQ` NOP on a `_NULL` variable. At this point, the callee pops the values of the arguments from the stack at step viii), by storing them in the inner variables and by decreasing the value of `m_ptr`. In this implementation, values are saved to its inner variables `Var_0_spiWrite` and `Var_1_spiWrite`, which are the two first variables of `spiWrite`. In LLVM-IR, the numbering of the inner variables in a function starts from `%1`. In our implementation, because we assume to start from `%0`, there is a one-element numbering gap between the function arguments and `alloca` variables (in this example `alloca` starts from `%3`). The callee function can, in general, implement other function calls (or for example a call to itself in case it is a recursive function), therefore it is fundamental to push `link_register` onto the stack besides `m_ptr`, step ix). By saving all the previous states in the stack the callee can then allocate elements in the stack according to its `alloca` variables, by using the offset address with respect to new callee `m_ptr` not to change the caller stack pointer. To address a stack element, the `alloca` implementation indeed needs to refer to the current function stack pointer and increment it with an offset calculated on the progressive enumeration of the `alloca` variables.

After execution of the function, step x) at the callee side, when returning, it is first necessary to pop the `link_register` from the stack and restore it in the dedicated variable, so that the callee can update the program counter as the last step. Depending on a void return or not, the callee at step xi) needs to push the value to be returned onto the stack, in this case, `Var_9_spiWrite`, and finally set the program counter to the value of `link_register` in step xii). At the caller side, it is just necessary to pop the return value, step v), decrement `m_ptr` by the number of `alloca` of the caller (in this case 1), and pop the old caller `m_ptr`, step vi). Now, the stack pointer is restored to the previous value before the function call and the caller stack variables are preserved.

Fig. 6 shows the generated assembly output (in CISC mode for the sake of brevity) for an example C code `foo.c` that writes a progressive number from 0 to 10 (i.e., `MAX`) to IOR, with all the I/O pins set as output. Observe that we have compiled the code by removing the definition of function `memcpy` normally present in `mOISC.c`. To show the usage of the previously introduced standard calling convention, we demand the writing of IOR to a minimal function `foo` that has a single integer argument. On the top left of the figure, an equivalent OISC mode code snippet for two instructions is shown for completeness. The `MCR` values in the data memory shown on the right are used to set the required machine mode only in OISC mode. After the mandatory definition of the machine registers, the compiler inserts an unconditional jump to the main routine which is assumed not to exit and to include an infinite `while(1)` cycle. Observe that `MCR` is 0 because in this example the machine boots in CISC mode. Thereafter, the compiler considers the LLVM-IR code and generates the assembly code in sequence, first for the function `foo` and later for the function `main`.

The function `foo`, whose entry point is defined by the

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

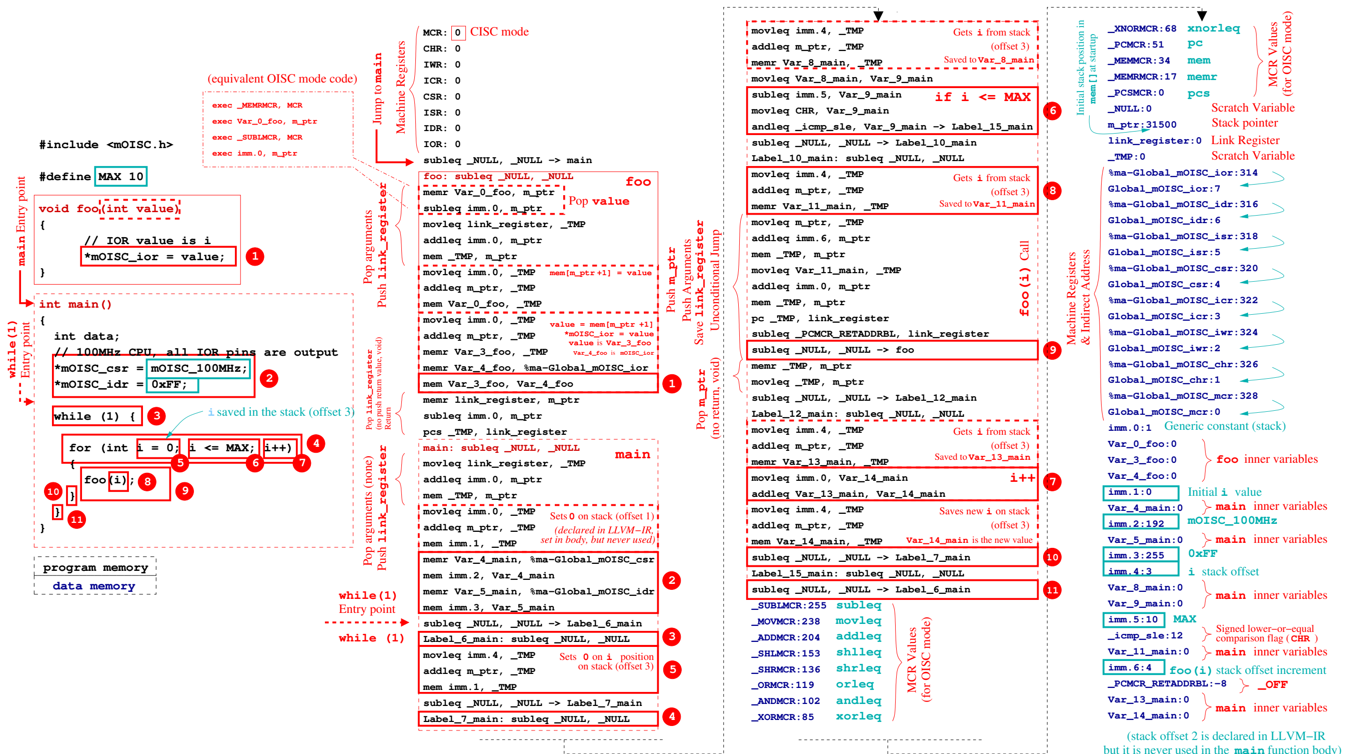


FIGURE 6. Example *m*OISC assembly with corresponding C code, generated using the basic LLVM-IR compiler (standard calling convention).

label `foo:`, first implements the steps vii) to xi) shown in Fig. 5. After the function start, the argument `value` is stored in a local function variable `Var_0_foo` that corresponds to a pop of the single argument from the stack. Next, following the calling convention steps, the `link_register` is pushed on the stack (`m_ptr` is incremented by 1, i.e., `imm.0`). These operations conclude the callee function definition in the scheme depicted in Fig. 5. Following the LLVM-IR code, the argument of the function is put on top of the stack, i.e., loaded and stored on a local variable whose address is defined using an `alloca` instruction, the first and only one of `foo`. Hence, the value of `Var_0_foo` is stored in the stack in position `m_ptr + 1`. Next, still following the LLVM-IR code generated by `clang`, the same value is taken from the stack and saved on a local variable `Var_3_foo`. Then, the address of `IOR` is accessed using a `memr` instruction that considers the indirect address of `Global_mOISC_ior` (prefix `%ma-`). This indirect variable stores the address of the memory cell whose value is the address of `IOR`. This redirection, which is applied for the mapping of LLVM-IR global variables, is necessary for mOISC, as no direct address extraction instruction is available, while memory addressing can be achieved only using double-depth addressing. In our toolchain the `mOISC.h` header file includes the definition of all machine registers as volatile integer pointers, therefore in the code, this redirection is applied to all machine registers. Next, the content of the local variable `Var_3_foo` is written on the address pointed by `Var_4_foo`, i.e., `IOR` (step ① in the figure). At this point the function can return to the caller, therefore implementing, the steps x), xi) and xii) defined in Fig. 5. In this specific case, no return value needs to be provided to the caller (the function is declared as `void`), therefore the code generator considers only step x) and xii).

After function start (label `main:`), function `main` pushes `link_register` onto the stack to implement the same steps of `foo`. Next, since this function never returns any value and implements an infinite loop, the compiler pushes a constant zero in the position `m_ptr + 1`, corresponding, in LLVM-IR, to the local variable `%1`, assumed to be in the stack. This stack variable and also the next one, i.e. `%2` (accessed in the stack at offset 1 and 2) are provided by LLVM in the intermediate representation but they are never used because the routine never returns. Next, step ②, the two assignment instructions in the C code referred to the setting of `CSR` and `IDR`, are implemented using the indirect addresses `%ma-Global_mOISC_csr` and `%ma-Global_mOISC_idr`, by exploiting a double-depth `mem` instruction with immediates `imm.2` and `imm.3` whose value is indeed 192 and 255, respectively.

At this point, the code implements an unconditional jump to the next basic block number 6, which encodes the infinite loop that has its entry point identified as label `Label_6_main`, step ③. The presence of such unconditional jump at this point of the code originates from the presence of a `br label %6` instruction in LLVM-IR immediately preceding label `6:` that our compiler translates without further optimization. Following the LLVM-IR out-

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

put, the integer variable `i` of the C code has been mapped to be on the stack, in particular at position `m_ptr + 3` (offsets 1 and 2 have been previously discussed). The value of `i` is initialized at the value of `imm.1`, i.e., `0` as per initialization of the `for` cycle in the C code, step ⑤. As the next step (hierarchically identified as ④ in the code because referring to a higher abstraction level), the compiler defines a further basic block labeled as `7` (labeled `Label_7_main`) which identifies the body of the `for` cycle. At this point of the code, the value of `i` needs to be checked to implement the termination condition of the `for` cycle. The value of `i` is indeed popped from the stack (offset 3, i.e., `imm.4`), saved in `Var_8_main`, and in turn copied to `Var_9_main` using `movleq`, to provide a temporary variable to implement the `for` condition.

In step ⑥, `Var_9_main` is compared to `imm.5` to implement `i <= MAX`. In our implementation, the selection scheme is the same used for direct ARM code translation (see Sec. IV-A2 for further details) that exploits the control flow of `ANDLEQ` and the comparison flags provided by the CPU in `CHR`. In this particular case, a constant `icmp_sle` is used to store the comparison flags that occur in the case of lower-or-equal conditions. First, `Var_9_main` is compared to `imm.5` using a `SUBLEQ` instruction. Then, `Var_9_main`, which will not be used anymore by definition of SSA, is overwritten with the value of `CHR`. If `i <= MAX` the system can continue to `Label_10_main` to implement the function call to `foo`. Otherwise, the CPU jumps to the basic block 15 (`Label_15_main`) to reiterate on `Label_6_main`, therefore on the `while(1)` loop, step ⑪.

The `foo` function call within `Label_10_main`, that follows the steps i)-iv) shown in Fig. 5, considers the local variable `Var_11_main` as temporary storage of the value of `i` that is extracted from the stack (offset `imm.4`). `Var_11_main` is then pushed onto the stack as argument `value`, step ⑧. The last step of the calling convention procedure implements the unconditional jump (step ⑨). Because `foo` returns `void`, the code that follows only pops `m_ptr` from the stack to restore the caller stack pointer according to steps v) and vi) of the calling convention. The last block 12 (labeled `Label_12_main`) implements the increment on `i`. As usual, the current value is first extracted from the stack and saved locally to `Var_13_main`. Next, a value of 1 is loaded in `Var_14_main` that is in turn added to `Var_13_main`, step ⑦. Finally, the new value of `Var_13_main` is written back onto the stack in the position corresponding to `i`. As increment is necessary only in case the `for` cycle needs to be iterated, the block concludes with an unconditional jump to `Label_7_main`, step ⑩.

2) Target Direct Translation

In case of direct translation from LLVM targets, in this implementation `x86`, `arm`, `mipsel` or `riscv`, invoked using the `-arch` argument, the complete LLVM front-end and back-end toolchain is utilized with both `clang` and `llc`. The *m*OISC compilation utility in this case reads and parses the assembly generated by LLVM, saved with the same extension of the target machine, to generate both OISC and CISC code. Translation is achieved using a set of primitives included in subfolders `/lib/arch`, a collection of files for each LLVM target. These primitives are a one-to-one direct translation from target instructions to *m*OISC assembly. For instance, given an ARM target, the simple instruction `sub %1, %2, %3`, where `%1` is the register that stores the result and `%2` and `%3` are the operand registers or immediates, is mapped as,

```
1    exec _MOVMCR, MCR
2    exec %2, _TMP
3    exec _SUBLMCR, MCR
4    exec %3, _TMP
5    exec _MOVMCR, MCR
6    exec _TMP, %1
7  _MOVMCR:    238
8  _SUBLMCR:   255
9  _TMP:       0
```

where lines 1–6 are the corresponding instructions in program memory and lines 7–9 are the required values that need to be appended (if not already present) in the data memory. Observe that thanks to the simplicity of *m*OISC, there is no difference when translating instructions in case of register-register, register-immediate, memory-register, or register-memory addressing because the *m*OISC memory organization is flat. This ARM instruction is implemented by first moving `%2` to a temporary register `_TMP` using a `MOVLEQ` machine mode, `_TMP` is updated in `SUBLEQ` mode with the difference `_TMP - %3`, i.e., `%2 - %3`, and the result is moved back to `%1` using `MOVLEQ`. To implement branches, multiple options are possible because the ISA implements control flow for all arithmetic and logic instructions. In this implementation, we have exploited the CHR flags and `ANDLEQ`. For instance, the ARM `ble` instruction is implemented as,

```
1    exec _ANDMCR, MCR
2    exec _LE_FLAG, _RESULT -> isNotLowerEq.%4
3    exec _SUBLMCR, MCR
4    exec _NULL, _NULL -> %1
5  isNotLowerEq.%4:    exec _SUBLMCR, MCR
6                      exec _NULL, _NULL
7  _ANDMCR:     102
8  _SUBLMCR:    255
9  _LE_FLAG:    12
10 _NULL:       0
```

where `_RESULT` is the content of CHR in the preceeding `cmp` instruction which implements a `SUBLEQ` between two comparison operands. Here, the code checks if `_RESULT` has equal and lower flags unset (i.e., on position `0x04` and `0x08` of CHR, overall `_LE_FLAG` = 12) by exploiting the control flow of `ANDLEQ`. If unset, the routine jumps to `isNotLowerEq.%4`, where `%4` is the current target assembly line, thus continuing execution. Otherwise, a `SUBLEQ` unconditional jump to `%1` is implemented using a `_NULL` variable. Observe that `mc.py` needs to generate unique names for each internal label that may be present in instruction translation, and here we have chosen to declare labels by embedding the current target assembly line number `%4` in our implementation.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

mc.py, by reading all the primitives at start-up time and by parsing the target architecture assembly, is then able to generate *m*OISC code although not globally optimized and providing lower performance compared to the target ISA. In a similar way with respect to the LLVM-IR compiler, translation ends up with an unlinked assembly that is linked with the internal linker module. To generate CISC mode assembly, the same flow of the LLVM-IR compilation case is applied by re-using the same internal functions, i.e., CISC remapper and Linker with PC remapper.

B. *m*OISC HARDWARE AUTOGENERATOR

We designed a mautogen.py utility that considers Quartus project VHDL files templates (software version at least 19.1) and auto-generates in a specific folder the complete processor with embedded binary and by considering only the effectively used subset of instructions of the *m*OISC ISA in the compiled software. The templates are organized in static and dynamic source files. The lib/autogen/static subdirectory comprises files that do not change, while /lib/autogen/rules includes files that need to be parsed and modified to generate the final code. mautogen.py, which accepts a specific IOR pin mapping with the argument -pincfg, first reads the OISC mode assembly with an ASM Reader module, it parses the dynamic files using a Template Parser, and finally, the File Processing module generates the Quartus project files for a Cyclone 10LP evaluation board. The ASM Reader module detects the data memory variables ending with MCR (therefore, those that define a valid machine mode) to understand the number of utilized modes used in the program. This information is passed to the Template Parser that reads the dynamic files, parses specific comments in the VHDL code and selects only the specific description parts that define the effectively used instructions in the hardware. Finally, the File Processing module copies the static files and the dynamic files with re-arranged code and pin configuration in a specific folder.

C. *m*OISC ASSEMBLER AND SIMULATOR

m.py includes both the *m*OISC assembler and a text-based interface simulator with the capability of generating VCD files for graphical simulations, normally supported by EDA graphical viewers. m.py considers the assembly code in both OISC and CISC modes to generate i) a corresponding memory initialization file fpga.mif to be used in the Quartus project to initialize internal RAM with FPGA firmware, ii) a binary file of the assembled program (sensor.bin in this example), iii) a text-based symbol file that identifies the address of each program variable or label (with syntax name @ value, where name is the symbol name and value is the address). For ease of implementation, we designed two separate assembler modules (both multi-pass) OISC mode assembler and CISC mode assembler to generate binary files because memory address differs in the two cases as well as the content of the text strings that need to be parsed.

The program enables two types of simulations, one that is text-based in which at every CPU cycle important variables specified by the argument -syms are printed on the screen, and another that simply aims at generating a VCD file to be processed for instance by gtkwave, an open-source software that enables graphical representation for both digital and mixed-signal simulations [31]. In this last case, it is mandatory to specify also the simulation time (-time argument) and optionally the storage of debug information, -debug. These, include the function names that are called while running the code in a text-based format. The simulator implements a behavioral Python description of the *m*OISC, with a back-annotated number of clock cycles per mode from the VHDL description. The program is read directly in the binary format generated by m.py. The IOR read events and the consequent ISR values following an IWR write event, detected during execution, are acquired from the standard input.

Fig. 7 shows a graphical output of gtkwave while running an example program sensor.c (see Sec. V for further details). m.py generates VCD data for both CPU machine registers, and based on the symbol file after compilation, it can provide debug information to check the operation of the CPU. In this example the cursor is placed corresponding to the function SPIWRITE (called by RH_RF95_INIT), and shows some of the internal data memory variables such as M_PTR and some internal variables of the MAIN function. IOR shows the four SPI signals involved in the communication with the RFM9x module. The VCD output feature provided by the simulator combined with gtkwave form a powerful verification tool, even for the design and the debug of the compiler.

V. PROOF-OF-CONCEPT MICROARCHITECTURE VALIDATION

Fig. 8(a) shows the test setup used to verify the correct operation of *m*OISC assuming a simple wireless telemetry application in which environmental temperature is read using a MAX3025 temperature sensor [32], and transmitted with an RFM9x LoRA chip [33]. To implement the set-up we took advantage of the Arduino connector available on the Cyclone 10LP development board to provide the necessary 3.3 V supply for both sensors and wireless transceiver. The IOR pins (0 to 7) in the VHDL description has been synthesized on pins B1, C2, F3, D1, G2, L14, G1, J2, the 50 MHz input clock is assigned to E1 and the reset signal RST is assigned to D9. The I^{2}C pins SCL and SDA of the MAX30205 are connected to IOR[7] and IOR[6], respectively, the SPI pins CS, MISO, CLK, and MOSI are connected to IOR[3-0], the RESET pin of the RFM9x is connected to IOR[4] and a LED indicator on the development board is connected to IOR[5]. To enable data reception from the RFM9x transceiver an Adafruit Feather 32u4 RFM9x board [34], programmed with the Arduino Integrated Development Environment (IDE), including the same transceiver, has been used as LoRA RX. The board outputs the received data in

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

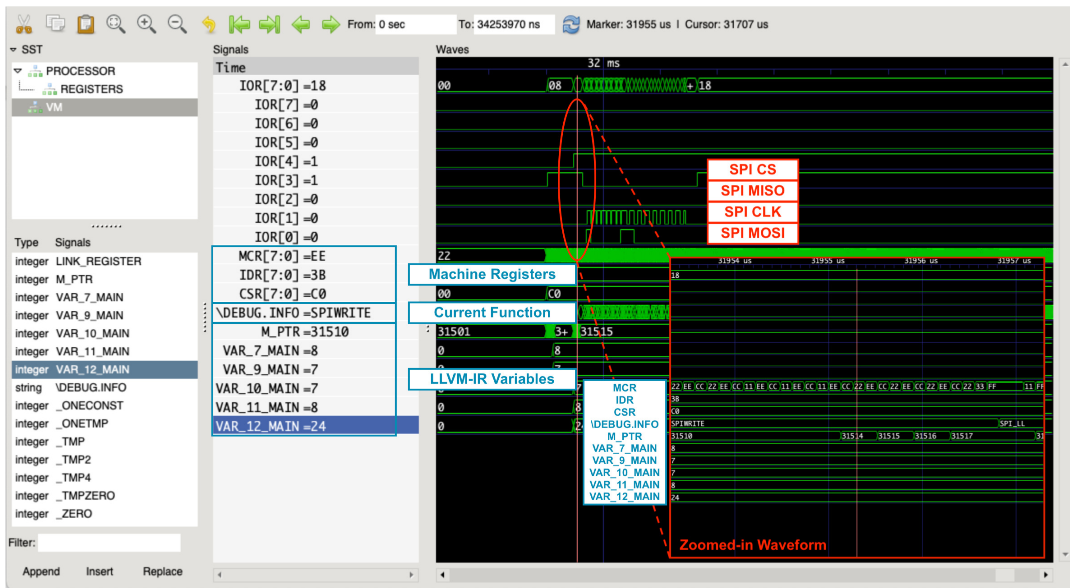
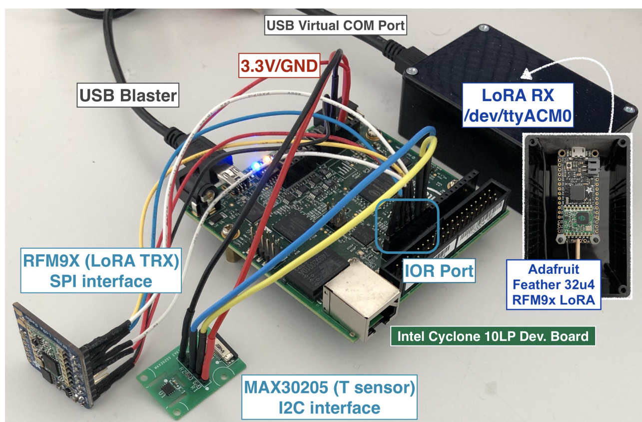
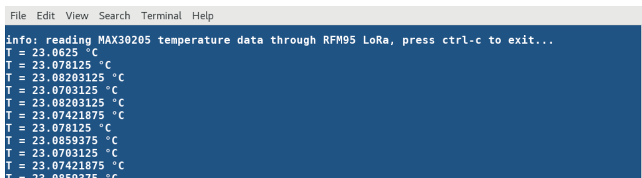


FIGURE 7. Example *m*OISC simulation output of the compiled bytecode from `sensor.c` (see Sec. V) viewed used `gtkwave`. The simulation includes also debug information.



(a)



(b)

FIGURE 8. **(a)** *m*OISC proof-of-concept microarchitecture validation setup using a Cyclone 10LP development board and two commercial chipsets, a RFM9x SPI transceiver and a MAX30205 body I^{2}C temperature sensor. To wirelessly receive temperature data, a separate system comprising a LoRA receiver (`LoRA RX`) with USB virtual communication port has been prototyped. **(b)** received data from *m*OISC read through the `LoRA RX` USB virtual COM port on `/dev/ttyACM0` using out custom Python script.

a two bytes binary format (integer and fractional part) on a virtual USB COM port, so that they can be processed and printed on the screen according to the MAX30205 specifications by a custom Python script. A typical temperature data output transmitted by *m*OSIC and received through the `LoRA RX` virtual COM port is given in Fig. 8(b). The IWR mechanism has been verified using the pushbuttons included in the Cyclone 10LP evaluation board with another custom program not shown here for the sake of brevity.

Lst. 2 shows the C code of the main function in the example program `sensor.c` used to verify system operation. Machine registers pointers are defined in the `lib/inc` sub-directory file `mOISC.c`, and are identified with suffix `mOISC` (for instance CSR address is given by `mOISC_csr`).

```
1  int main()
2  {
3    // I2C buffer
4    int i2cbuf[2] = {0, 0}; int length = 2;
5    // sets CPU speed at 100MHz
6    *mOISC_csr = mOISC_100MHz
7    // initializes SPI, LED and I2C ports
8    spi_led_init();
9    i2c_init();
10   // RFM9x RESET pulse, set
11   *mOISC_ior = 0xEF & *mOISC_ior;
12   // waits for 10 idle cycles
13   delay(10);
14   // unset
15   *mOISC_ior = 0x10 | *mOISC_ior;
16   // RFM9x initialization, returns 1 if done
17   if (RH_RF95_init() == 0) {
18     // if not initialized, wait forever
19     while(1);
20   }
21   // main loop
22   while(1)
23   {
24     // reads MAX30205
25     MAX30205_read(i2cbuf);
26     // sends data to RFM9x
27     RH_RF95_send(i2cbuf, length);
28     // sets CPU speed to 10kHz
29     *mOISC_csr = mOISC_10kHz;
```


This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

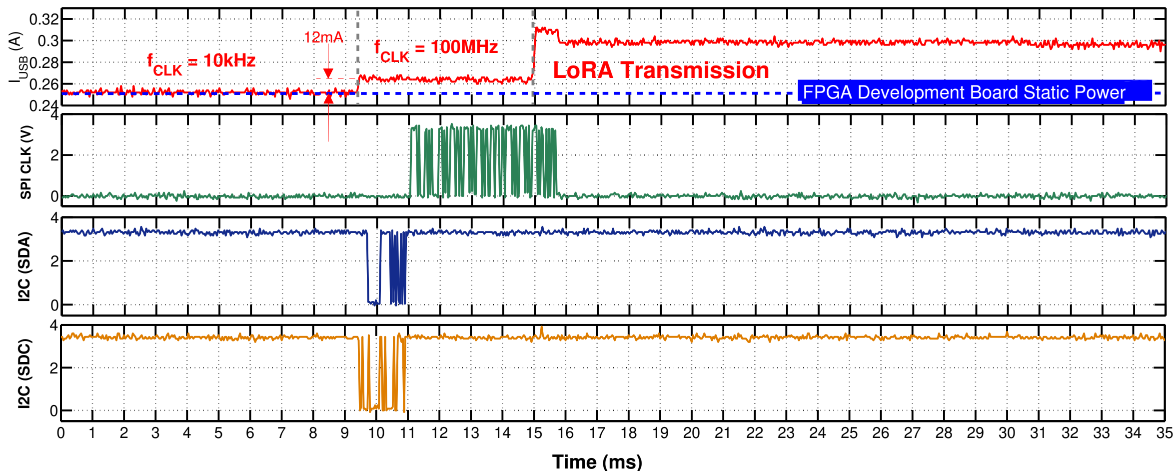


FIGURE 9. Measured current consumption of the USB port of the complete Cyclone 10LP evaluation board while running `sensor.c`. The LoRA transceiver and the temperature sensor take the supply voltage from the evaluation board.

```
30      // blinks LED on IOR[5]
31      *mOISC_ior = 0x20 | *mOISC_ior;
32      delay(1);
33      *mOISC_ior = 0xDF & *mOISC_ior;
34      delay(1);
35      // sets CPU speed to 100MHz
36      *mOISC_csr = mOISC_100MHz;
37    }
38  }
```

Listing 2. `sensor.c` main function code snippet.

The program, after initialization of both SPI and I^{2}C (that are implemented using a shared global variable to determine the I/O direction of the IOR port), runs the following operations: it initializes the LoRA transceiver and enters an infinite loop, in which, i) at high clock speed (100 MHz), it reads the current temperature through I^{2}C from the MAX30205, ii) sends this data as through SPI to the RFM9x transceiver in broadcast mode, iii) slows down the CPU to 10 kHz to blink a LED on `IOR[5]`, and iv) resets the CPU speed to 100 MHz and repeats. The function `delay` implements an empty `for` cycle. This very simple program demonstrates a very basic solution to a wireless telemetry problem of remote temperature sensing, and through the infinite loop, makes it possible to measure the performance of the CPU while running code obtained from LLVM-IR or the assembly of different target machines.

A. MICROARCHITECTURE POWER CONSUMPTION

Fig. 9 shows the current consumption of the complete Cyclone 10LP evaluation board with synthesized *m*OISC while running `sensor.c`. To measure power consumption we have built a custom USB cable with exposed 5 V supply wires to enable current measurements using a Tektronix TCP0030 current probe and an MSO4104 oscilloscope. As it is not possible to measure the leakage power of the FPGA and thus understanding the contribution that goes directly to the processor, the only measurement possible in these conditions regards dynamic power. Measurements show a $I_{\mathrm{USB}} = 12$ mA absorption from the 5 V USB port supply, for a 60 mW power consumption, while running at 100 MHz. At 10 kHz the dynamic power consumption of the CPU is reduced by a factor of 10^4 (estimated on the order of 60 mW/10^4=6 μW) and it is not possible, using our current probes, to appreciate a variation with respect to the board static power. Observe the peak current consumption increases after the activation of the LoRA transceiver, which is designed to consume duty-cycled current, i.e., only during packet transmission. Due to the lack of available ports in the oscilloscope, we have shown only one out of four SPI signals (`SPI CLK`). The digital signals waveform sampling rate is dictated by the current probe, hence, the I^{2}C `SCL` and `SDA` signals are downsampled.

B. TRANSLATED/COMPILED CODE PERFORMANCE

Fig. 10 and Fig. 11 show the compiled code size and the duration of the main loop shown in Lst. 2 of our sample program `sensor.c`, translated from all the supported target ISA and compiled using `clang` LLVM-IR. The execution time is measured using a DSO9404A oscilloscope, by reading the SPI clock cycle time. The worst results both in terms of code size and execution are obtained with MIPS and RISC-V intermediate microcode. This is due, in general, to the higher number of assembly instructions emitted within the same LLVM basic block. A basic block is a container of instructions that execute sequentially [30], [35]. For MIPS and RISC-V each operand in any assembly line is conceived to save data at address `c` based on address `a` and `b` notwithstanding *m*OISC naturally overwrites data at address `b` as a function of data in `a` and `b`. Observe that direct translation from commercial targets is typically implemented line by line in a *flat* manner by `mc.py`. By running translation with these assumptions, it is clear that the emitted code is larger and has lower performance compared to the other targets.

When code is compiled from LLVM-IR, occupation is still high, due to a lack of optimization and direct LLVM-IR

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

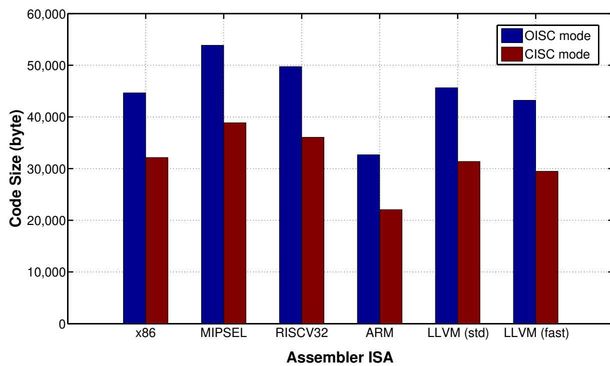


FIGURE 10. *m*OISC proof-of-concept compiled code size assuming different assembly ISA obtained using LLVM assuming same program `sensor.c`, in both CISC and OISC mode.

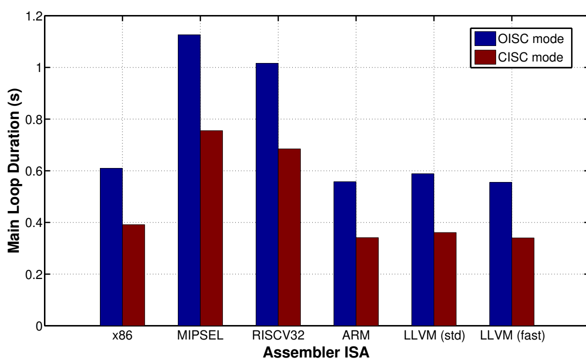


FIGURE 11. *m*OISC proof-of-concept execution time of the `sensor.c` main loop of the programs whose binary size in reported in Fig. 10.

register translation. However, LLVM-IR compilation leads to the best performance in terms of execution time thanks to the possibility to handle code generation from a higher abstraction level language. For ARM and x86 target translation, the obtained speed is higher, because in both cases the number of assembly instructions within each LLVM basic block is lower compared to MIPS and RISC-V, notwithstanding the large number of operands per instruction. ARM and x86, indeed, enable to compactly combine multiple registers in single instructions and calculate addresses more easily with embedded offsets (see for instance `movl` in x86 or `ldr`/`str` in ARM ISA). Therefore, the translator, that runs a *flat* line-by-line translation, is capable of implementing a more optimized code generation.

The CISC mode increases the memory occupation per instruction (4 addresses versus 3 per instruction) but in terms of efficiency and overall code size, it is advantageous. In our simple software translator indeed, the generated OISC code keeps alternating an MCR write and the execution of the selected instruction, thus duplicating program memory occupation and execution time because one instruction is always wasted for setting MCR. The CISC mode code size is scaled by an approximate factor of 2/3 w.r.t. those in OISC mode.

C. RESOURCE OCCUPATION

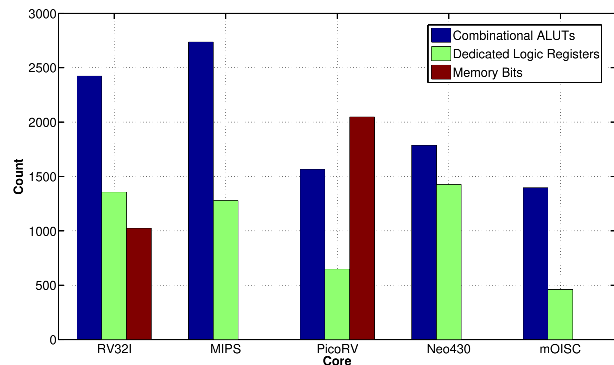


FIGURE 12. *m*OISC proof-of-concept synthesis results (full featured processor with all instructions) against other open-source cores for 32 bit and 16 bit ISA assuming the same Cyclone 10LP FPGA target and the same 50 MHz clock.

Fig. 12 shows the area occupation of *m*OISC compared to open-source CPU for similar application domains [24], [36], [37]. We have synthesized all cores on the same Cyclone 10LP FPGA with the same 50 MHz clock in order to maintain a fair comparison, and by excluding the external memory and the fabric FPGA JTAG interface from the count for all processors. For *m*OISC we have included in the count the logic required by the PLL. For the PicoRV core and RV32I, we obtained non-zero memory bit elements (1024 and 2048 bit of M9K blocks) because register files are synthesized using the FPGA RAM memory, hence it is fair to include them as part of the core. We have synthesized *m*OISC with all possible instructions. Notwithstanding that our proof-of-concept disregards area and performance optimization and implements a simple and underperforming FSM, it remains the lower area count processor. Resource occupation remains significantly lower compared to other pipelined non-bus based 32 bit microarchitectures not listed here, based on RV32I instruction sets. For instance, the RISC processor in [38], although implemented on Spartan FPGA occupies 5578 LUTs and 1073 flip flops. *m*OISC resource occupation is lower compared to an open-source implementation of the 16 bit MSP430 (Neo430). Further resource occupation reduction is possible, especially by utilizing a bus for interconnecting the core with the memory and registers. With an LLVM-IR compilation, the resulting assembly uses all instructions besides `XORLEQ` and `XNORLEQ`. By running synthesis with `mautogen.py`, therefore excluding these two unused instructions from the `ALU`, the number of combinational ALUTs becomes 1393 with the same amount of logic registers, thus saving 4 ALUTs compared to the full-featured processor. Better results may be possible with ASIC implementation. The amount of logic utilized, indeed, can be more finely tuned compared to using the fixed logic fabric of the FPGA. However, the synthesized silicon area severely

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

depends on available digital cells and this aspect deserves separate investigation [39].

VI. ISA PERFORMANCE AND DISCUSSION

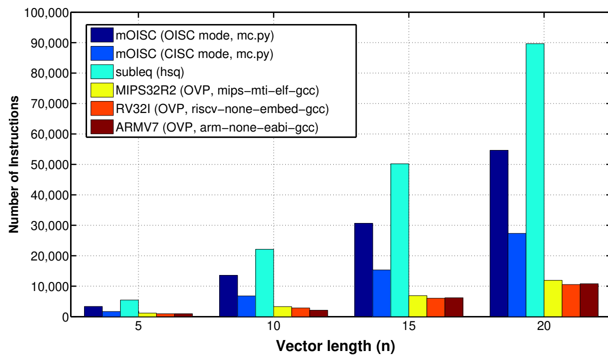


FIGURE 13. Number of executed instructions to run the same Bubble Sort algorithm on an integer vector `V` for different length n (5, 10, 15 and 20), for an *m*OISC, a `subleq` OISC, and for known MIPS, RISC-V, and ARM architectures, with corresponding compilers. The *m*OISC assembly is generated using our basic LLVM-IR compiler.

To compare the performance of the *m*OISC ISA combined with our compilation toolchain to other known architectures, we have run benchmark simulations assuming a simple Bubble Sort algorithm applied on an integer vector of n elements, that needs to be fully re-ordered in ascending order (worst case, initialized as ordered in descending order). We have decided not to consider the number of clock cycles required to complete an instruction so that the performance of the ISA (compiler included) can be compared independently from the corresponding microarchitecture. We have patched the open-source implementation of Higher Subleq (see [1], [9], downloadable at [10]) to print out the number of execution cycles once the execution finishes (i.e., defining an `int cycle`, incremented in the simulation loop and printed for `ip < 0`, namespace `emulator`, routine `sqemulatei`). Observe that Higher Subleq implements stack with self-modifying code, and includes specific optimizations for `subleq`. For the evaluation of commercial ISA, we have used Open Virtual Platform sim (OVPsim) which provides several open-source models and application programming interfaces for simulation of known architectures [40]. For *m*OISC, we have compiled the code using the LLVM-IR compiler with standard calling convention. For MIPS, RISC-V, and ARM the compilation toolchain provided with OVPsim have been used, in particular, `mips-mti-elf-gcc`, `riscv-none-embed-gcc` and `arm-none-eabi-gcc`, respectively. The compiled code includes also specific calls to intercept program completion, and to set machine registers for *m*OISC. Their impact is included in our results but it is not significant in the overall executed instructions count.

Fig. 13 shows the number of instructions required by each specific ISA to complete the bubble sort algorithm as a function of the number of elements to be re-ordered n. Results confirm that the worst-case Bubble Sort complexity, as expected, goes as $\mathrm{O}(n^2)$, for all cases, irrespective of the ISA. Compared to a pure `subleq` model, *m*OISC in OISC mode runs 1.6 × faster, and 3.3 × in CISC mode. Other more complex and more performing ISA complete the execution with a significantly lower number of instructions, i.e., ∼2.3 ×, ∼2.6 ×, and ∼2.6 × for MIPS, RISC-V, and ARM, respectively. The results in terms of performance for CISC mode are promising: even with such a limited instruction set and using a simple compilation toolchain the obtained performance in terms of coded instructions is roughly a factor two larger compared to MIPS. Observe that the ARM, RISC-V, and MIPS ISA instructions compactly encode also arithmetics with offset calculations required for fast accessing of the stack, while *m*OISC includes only generic double-depth addressing and offsets need to be explicitly calculated using other arithmetic instructions.

Assuming another sample program that runs the bubble sort algorithm on a 32 element integer vector, compared to an ARM ATSAMD21, *m*OISC runs 57× slower, assuming the same 50 MHz clock (OISC mode), here accounting for the real execution time and not only the number of instructions. This is undoubtedly due, as previously introduced, to the limited instruction set, and, most importantly, to the slow fetch and execute phases of our proof-of-concept implementation. In OISC mode, indeed, the current prototype cycle time lasts 10 clock cycles for `PCS` and machine register writes (except for IWR), 11 clock cycles for any `LEQ` or `PC` instruction, 12 clock cycles to write IWR, and 3 clock cycles to exit from interrupt. `MEMR` and `MEM` require 11 clock cycles to be completed. In CISC mode two additional clock cycles are required to complete an instruction except when exiting an interrupt block. This performance is aligned with the old but still used Intel MCS8051 CPU, which requires 12 clock cycles to run an instruction [41]. By synthesizing the 8 bit 8051 core in [42] on the same Cyclone 10LP FPGA (CPU only), we obtain 982 ALUTs, 346 dedicated logic registers, 4096 memory bits, and a `9x9` DSP element. Adding up all the contributions, *m*OISC, notwithstanding working at 16 bit, provides lower resource occupation.

The slow execution speed of *m*OISC is intrinsic in the architecture of the machine that assumes a pure von Neumann scheme with flat absolute addressing, unpacked instructions, and no register file. The presence of very fast access storage, such as a register file, would speed up execution speed, thus leaving more time-consuming memory access instructions, executed less frequently. By using an aggressive pipelining (which results in an increased number of registers at implementation-level) we can ideally reduce the average number of clock cycles to 4 per instruction. The other bottleneck refers to the single-port RAM that is used for the implementation of the central memory. By using a dual-port RAM as suggested in [1], we can then speed up instruction fetch, thus reducing by an additional factor 2 the number of clock cycles. Moreover, another possible improvement regards the use of a Harvard architecture in conjunction to

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

pipeline. However, increasing performance may impact our research objective of reducing silicon area and number of resources for environmental purposes. Moreover, the presence of a pipeline impacts on real-time execution. As a matter of fact, slow-speed CPUs can still find relevancy in some applications in which physical quantities to be observed are somehow not too fast, e.g., in smart agriculture applications, provided that circuits can operate at reduced energy and aggressive duty cycling. Our development could go in favor of such application domain in which normally electrical energy can be extracted from solar cells, thus making our blocking interrupt mechanism a potentially useful functionality.

To fully take advantage of our *m*OISC architecture (especially the various control flow options made available by the ISA), therefore minimizing the number of MCR writes, it is necessary to elaborate on a more sophisticated compiler or alternatively integrate the *m*OISC ISA into LLVM targets. Given the flexibility of *m*OISC, another possible improvement point regards instruction parametrization. In the current implementation, the MCR value is used as an index that selects the instruction to be executed among a fixed instruction set; we could, instead, interpret the MCR value, extended e.g. to the full 2 byte memory width, to compactly define machine mode details. We can consider the synthesis of ad-hoc modes by combining the already available arithmetic and logic resources and outputs of the ALU, thus flexibly re-using them to synthesize both control and data flow. For instance, given a hardware comparator we can re-use it to implement also jump if bigger or equal, or given arithmetic and logic hardware blocks, we can route them and combine their output to implement more complex math. This, *inter alia*, goes in favor of a re-engineering of the control unit for a more efficient implementation compared to a trivial FSM.

Cross-sectional approaches in the design of computing systems are highly demanded because, so far, processor design has been focused on the minimization of operational energy consumption, while carbon emission continues to grow due to hardware manufacturing and infrastructure [11]. In this respect, the referenced work states that low footprint circuit design is a potential opportunity for the reduction of carbon emissions. In terms of architecture design, judicious provisioning of the resources, hardware down-scaling, and the incorporation of ad-hoc hardware modules can make a difference in CO_2 emissions. The reduced resource occupation of *m*OISC and the possibility to synthesize the hardware based on the minimal set of actually used arithmetic and logic operations goes in favor of the aforementioned directions. These potential environmental impact advantages need to be demonstrated through silicon implementation.

VII. CONCLUSION

We have presented an ISA, with associated proof-of-concept microarchitecture, based on the extension of the minimalistic OISC approach towards a practical implementation for use in microcontroller applications. The proof-of-concept microarchitecture, even without encompassing specific area constraints, on a Cyclone 10LP FPGA achieves lower resource occupation compared to area efficient implementations of open-source 16 and 32 bit microprocessors. Although its performance is limited due to its intrinsic minimalism, this CPU has been proven to sustain low complexity wireless telemetry applications. The ISA can be easily extended to 32 bit to support a large memory capacity and thus larger programs. We have implemented a simple assembly translator from known ISA and a basic compiler from LLVM-IR that has been demonstrated to work effectively for basic programs. This work poses the basis for the devising of other ISA by considering the co-design of compiler and hardware towards more aggressive area minimization and lower complexity to favor, possibly, sustainability in silicon implementation.

ACKNOWLEDGMENT

The author would like to thank Claudio Lorini, Electronic Design Laboratory, for constructive discussions and support.

REFERENCES

[1] O. Mazonka and A. Kolodin, “A Simple Multi-Processor Computer Based on Subleq,” in *arXiv Distributed, Parallel, and Cluster Computing*, Jun. 2011, pp. 1–24.

[2] F. Mavaddat and B. Parhami, “URISC: The Ultimate Reduced Instruction Set Computer,” *Int. Journal on Electrical Engineering Education*, vol. 25, pp. 327–334, 1988.

[3] O. Mazonka, N. G. Tsoutsos, and M. Maniatakos, “Cryptoleq: A Heterogeneous Abstract Machine for Encrypted and Unencrypted Computation,” *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 9, pp. 2123–2138, 2016.

[4] A. Yildiz, S. Gören, H. F. Ugurdag, B. Aktemur, and T. Akdogan, “Crucial Topics in Computer Architecture Education and a Survey of Textbooks and Papers,” *International Journal of Computer Science*, vol. 43, no. 3, pp. 237–252, Aug. 2020.

[5] P. J. Nürnberg, U. K. Wiil, and D. L. Hicks, “A Grand Unified Theory for Structural Computing,” in *Metainformatics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–16.

[6] O. Mazonka, “Bit Copying – The Ultimate Computational Simplicity,” *Complex Systems Journal*, vol. 19, no. 3, 2011.

[7] P. Jääskeläinen, A. Tervo, G. Payá Vayá, T. Viitanen, N. Behmann, J. Takala, and H. Blume, “Transport-Triggered Soft Cores,” in *IEEE International Parallel and Distributed Processing Symposium Workshops*, 2018, pp. 83–90.

[8] “MaxQ Family User’s Guide”, Accessed June 25, 2021. [Online]. Available: https://pdfserv.maximintegrated.com/en/an/AN4811.pdf

[9] “Higher Subleq – typeless simplified C-like language which compiles into Subleq”, Accessed June 25, 2021. [Online]. Available: https://esolangs.org/wiki/Higher_Subleq

[10] “8l/hsq”, Accessed June 25, 2021. [Online]. Available: https://github.com/8l/hsq

[11] U. Gupta, Y. G. Kim, S. Lee, J. Tse, H.-H. S. Lee, G.-Y. Wei, D. Brooks, and C.-J. Wu, “Chasing Carbon: The Elusive Environmental Footprint of Computing,” 2020.

[12] N. Sakamoto, T. Ahmed, J. Anderson, and Y. Hara-Azumi, “Subleq $_{\ominus}$: An Area-Efficient Two-Instruction-Set Computer,” *IEEE Embedded Systems Letters*, vol. 9, no. 2, pp. 33–36, 2017.

[13] D. Bol, J. De Vos, F. Botman, G. de Streel, S. Bernard, D. Flandre, and J. Legat, “Green SoCs for a sustainable Internet-of-Things,” in *IEEE Faible Tension Faible Consommation*, 2013, pp. 1–4.

[14] E. Brunvand, D. Kline, and A. K. Jones, “Dark Silicon Considered Harmful: A Case for Truly Green Computing,” in *Ninth International Green and Sustainable Computing Conference (IGSC)*, 2018, pp. 1–8.

[15] T. G. Gutowski, M. S. Branham, J. B. Dahmus, A. J. Jones, A. Thiriez, and D. P. Sekulic, “Thermodynamic Analysis of Resources Used in Manufacturing Processes,” *Environmental Science & Technology*, vol. 43, no. 5, pp. 1584–1590, 2009.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3104150, IEEE Access

[16] "The Environmental Footprint of Logic CMOS Technologies", Accessed June 25, 2021. [Online]. Available: https://www.imec-int.com/en/articles/environmental-footprint-logic-cmos-technologies

[17] M. Badaroglu, J. Xu, J. Zhu, D. Yang, J. Bao, S. Song, P. Feng, R. Ritzenthaler, H. Mertens, G. Eneman, N. Horiguchi, J. Smith, S. Datta, D. Kohen, P. Chan, K. Chen, and P. R. C. Chidambaram, "PPAC Scaling Enablement for 5nm Mobile SoC Technology," in *European Solid-State Device Research Conference*, 2017, pp. 240–243.

[18] D. Kline, N. Parshook, A. Johnson, J. E. Stine, W. Stanchina, E. Brunvand, and A. K. Jones, "Sustainable IC design and fabrication," in *International Green and Sustainable Computing Conference*, 2017, pp. 1–8.

[19] W. F. Gilreath and P. A. Laplante, *Computer Architecture: A Minimalist Perspective*. Springer, Boston, MA, 2003.

[20] J. C. Furgal and C. U. Lenora, "Green Routes to Silicon-based Materials and Their Environmental Implications," *Physical Sciences Reviews*, vol. 5, no. 1, p. 24, Oct. 2019.

[21] "The NEORV32 RISC-V Processor", Accessed June 25, 2021. [Online]. Available: https://github.com/stnolting/neorv32

[22] S. Bora and R. Paily, "A High-Performance Core Micro-Architecture Based on RISC-V ISA for Low Power Applications," *IEEE Transactions on Circuits and Systems II: Express Briefs*, pp. 1–1, 2020.

[23] N. M. Qui, C. H. Lin, and P. Chen, "Design and Implementation of a 256-Bit RISC-V-Based Dynamically Scheduled Very Long Instruction Word on FPGA," *IEEE Access*, vol. 8, pp. 172 996–173 007, 2020.

[24] "PicoRV32 – A Size-Optimized RISC-V CPU", Accessed June 25, 2021. [Online]. Available: https://github.com/cliffordwolf/picorv32

[25] "mOISC-dRISC", Accessed June 25, 2021. [Online]. Available: https://github.com/MarcoCrepaldi-iit/mOISC-dRISC

[26] L. Null and J. Lobur, *The Essentials of Computer Organization and Architecture*. John & Bartlett Learning, 2010.

[27] "Cyclone 10LP Evaluation Kit", Accessed June 25, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/cyclone-10-lp-evaluation-kit.html

[28] "The LLVM Compiler Infrastructure", Accessed June 25, 2021. [Online]. Available: https://llvm.org

[29] J. See, W. Lee, K. Mok, and H. Goh, "Development of LLVM Compilation Toolchain for IoT Processor Targeting Wireless Measurement Applications," in *IEEE International Conference on Smart Instrumentation, Measurement and Application*, 2017, pp. 1–4.

[30] K. Shigenobu, K. Ootsu, T. Ohkawa, and T. Yokota, "A Translation Method of ARM Machine Code to LLVM-IR for Binary Code Parallelization and Optimization," in *International Symposium on Computing and Networking (CANDAR)*, 2017, pp. 575–579.

[31] "GTKWave", Accessed June 25, 2021. [Online]. Available: http://gtkwave.sourceforge.net

[32] "MAX30205 Evaluation Kit", Accessed June 25, 2021. [Online]. Available: https://www.maximintegrated.com/en/products/sensors/MAX30205EVSYS.html\#product-details

[33] "Adafruit RFM9x", Accessed June 25, 2021. [Online]. Available: https://learn.adafruit.com/adafruit-rfm69hcw-and-rfm96-rfm95-rfm98-lora-packet-padio-breakouts

[34] "Adafruit Feather 32u4 with LoRa Radio Module", Accessed June 25, 2021. [Online]. Available: https://learn.adafruit.com/adafruit-feather-32u4-radio-with-lora-radio-module

[35] "LLVM Basic Block", Accessed June 25, 2021. [Online]. Available: https://llvm.org/doxygen/classllvm_1_1BasicBlock.html

[36] "HF-RISC SoC", Accessed June 25, 2021. [Online]. Available: https://github.com/sjohann81/hf-risc

[37] "A very small msp430-compatible customizable soft-core microcontroller-like processor system written in platform-independent VHDL", Accessed June 25, 2021. [Online]. Available: https://github.com/stnolting/neo430

[38] D. K. Dennis, A. Priyam, S. S. Virk, S. Agrawal, T. Sharma, A. Mondal, and K. C. Ray, "Single cycle RISC-V micro architecture processor and its FPGA prototype," in *International Symposium on Embedded Computing and System Design (ISED)*, 2017, pp. 1–5.

[39] A. Boutros and V. Betz, "FPGA Architecture: Principles and Progression," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, 2021.

[40] "Open Virtual Platforms (OVPsim)", Accessed June 25, 2021. [Online]. Available: https://www.ovpworld.org

[41] "MCS 51 Microcontroller Family Users' Manual", Accessed June 25, 2021. [Online]. Available: http://web.mit.edu/6.115/www/document/8051.pdf

[42] "Yet another free 8051 FPGA core", Accessed June 25, 2021. [Online]. Available: https://github.com/jaruiz/light52

MARCO CREPALDI (M'9) received the engineering degree (summa cum laude) and the Ph.D. in electronic engineering from the Politecnico di Torino (Polito), Turin, Italy, in 2005 and 2009, respectively. During 2008 he was a Visiting Scholar at the Electrical Engineering Department of Columbia University in the City of New York. After the Ph.D., he worked as a Postdoc at the VLSI-Lab, Electrical Engineering department, PoliTo, and then as a Postdoc at the former Istituto Italiano di Tecnologia@PoliTo Center for Space Human Robotics (IIT-CSHR). He is now the coordinator of the Electronics Design Lab (edl.iit.it) at the IIT Center for Human Technologies in Genova. His scientific activity regards the development of all-digital Impulse-Radio Ultra-Wide Band (IR-UWB) systems and electronic systems design. He is author and co-author of more than 90 publications and two international patents.

ANDREA MERELLO received the Computer Science degree (summa cum laude) from Universitá di Genova, Italy in 2008. He works as software engineer at the Istituto Italiano di Tecnologia, Electronic Design Lab since 2008. His main activities regard software development in the field of Linux drivers, firmware for bare-metal and ultra-low power electronics, motor control, CAN bus and wireless communication. He contributed to the development of several open-source projects, including the Linux kernel with several patches. He designed electronics boards for experimental and scientific setups and several firmware and software for custom modules in the field of wireless systems, robotics, and miscellaneous systems. He co-authored one conference, one international journal paper and one patent.

MIRCO DI SALVO received the Computer Engineering degree at Universitá di Genova, Italy in 2011. He worked for automotive, telecommunication and aerospace industries before joining the Istituto Italiano di Tecnologia in 2014. He first worked for ReHab technologies and in 2017 he joined the Electronic Design Laboratory. His current activities regard the development of control software for different types of robots and firmware for motor control for robotic applications, radio tranceivers, experimental medical devices and smart sensors. His technical interests regard parallel computing architectures and 3D graphics.

• • •

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/