IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 30, 2021, accepted November 3, 2021, date of publication November 16, 2021, date of current version November 24, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3128382

An Abstract Machine Approach to Preserving Digital Information

IVAR RUMMELHOFF[1], ELADIO GUTIÉRREZ[2], THOR KRISTOFFERSEN[1], OLE LIABØ[3], BJARTE M. ØSTVOLD[1], OSCAR PLATA[2], AND SERGIO ROMERO[2]
[1]Norwegian Computing Center, 0373 Oslo, Norway
[2]Department of Computer Architecture, University of Malaga, 29071 Malaga, Spain
[3]Piql AS, 3045 Drammen, Norway

Corresponding author: Bjarte M. Østvold (bjarte@nr.no)

This work was supported in part by EUROSTARS under Project E!12494, and in part by the Regionale forskningsfond (RFF) Oslofjordfondet under Project 285516.

ABSTRACT Preserving digital information for a very long time is difficult even when using a durable passive storage medium such as photographic film stored under the right conditions. On film one can combine analog descriptions, that is, visual and thus human-readable text and diagrams, with encoded digital information. After hundreds of years, however, the formats used to represent and encode this information may have been forgotten, and any surviving source code may not simply be compiled and run. Explaining how to interpret data stored in a complex format runs the risks both of errors made today and of future misunderstandings. We present a solution based on (1) a very simple abstract machine, (2) independent, technology-neutral descriptions of the machine, preserved in analog form and aimed at future programmers and mathematicians, and (3) a C compiler targeting this machine. Currently, our toolset supports storing and retrieving data in the formats JPEG, TIFF and PDF/A, but other formats can be easily be added by adapting existing C programs for processing these formats. Binaries for the abstract machine are preserved alongside the digital information and the machine descriptions so that future generations can decode and present the information simply by implementing this machine.

INDEX TERMS Formal specifications, data storage systems, programming, codecs, information representation, computer languages.

I. INTRODUCTION

Imagine that you are a future historian, alive 500 years from now. You come across a durable digital storage medium that you believe contains important historical information dating back hundreds of years. The storage technology is no longer in use and it is not described by your available sources, and thus you do not know how to extract information from it. However, those who made the medium and stored data on it also provided an analog description, for example, engraved on a metal plaque.[1] This description can be read directly, explaining to you the procedures to read data from the medium and to interpret the data.

We assume the existence of an adequate storage medium. Our approach addresses how to preserve formatted data on the medium and how to ensure that it can be retrieved in the future. The toolset that we describe directly supports preserving data in certain formats and adding support for other formats straightforward. Future retrieval of the data requires doing some engineering work outlined in the description, but the description is generic and need not be changed when adding new formats. Our approach is independent of the type of storage medium, except that it must be possible to store both analog descriptions and digital data.

A. IMMEDIATE QUESTIONS

We introduce our approach by answering some questions.

Is there a medium available today that can reliably hold data for 500 years? Yes, one such storage medium is discussed in Section VII-C.

Do you preserve the original files and metadata? Yes, there is no data conversion when producing the storage instances, and the abstract machine can be used for extracting the original files in addition to presenting their contents.

The associate editor coordinating the review of this manuscript and approving it for publication was Yang Liu.

[1]Such plaques were used for this purpose aboard the two Pioneer spacecrafts on their journeys outside the Solar system.

What is the cost of preparing data for storage and producing the machine descriptions? Since we store the original files, data in the supported formats have zero production cost, and the storage cost is proportional to size. We currently support the formats JPEG, TIFF and PDF/A. Additional data formats require C code to render the data to the output devices of our abstract machine. The human readable descriptions are identical for every storage instance and thus have zero production cost, but incur a constant storage cost for the analog medium.

Will future historians and their helpers be able to understand the machine descriptions? We only assume that they will understand texts from our time that use the Latin alphabet, Arabic numerals and rudimentary English, and that are self-contained except for very basic technological concepts.

Will they have the resources to implement this machine? A basic implementation requires very little effort. Moreover, the descriptions will be the same for all storage instances. Thus, the cost can be amortized across all the instances they have available.

For images and documents, why not just store them as analog pictures on a visual medium such as film or paper? Analog, unencoded storage supports neither error correction nor compression, and the original files will be lost. Moreover, many types of digital information do not have any obvious visual representation.

B. OUR APPROACH AND CONTRIBUTIONS

Our approach to long-term preservation of digital information is realized as a toolset which includes a C compiler, an abstract machine targeted by the compiler, and descriptions explaining the machine to future readers. The process is automatic for information in supported formats, currently JPEG, TIFF and PDF/A. Adding support for other formats will usually involve porting an existing C program to this machine (and its I/O devices).

The use of an abstract machine to be implemented in the future allows us to sidestep two problems of long-term preservation of digital information: (1) the limited lifespan of the software used to interpret the stored information, and (2) the need to explain the media and storage formats themselves to future readers. The price is the future effort needed to understand and implement the machine. In other words, machine code serves as a universal format. For example, the most complex format we currently support, PDF, is described in a 968-page document, which would require a significant effort to understand and implement. By contrast, our approach enables competent programmers to render PDF documents after following a series of simple steps in a 17-page document. There is no need to understand PDF or any other of the supported formats.

Having a C compiler that targets our abstract machine was essential, beyond supporting preservation of formats defined in C code: We implemented key features as either C with embedded assembly code or by porting C libraries to the machine. A floating point library was required for PDF/A support and also for decoding film frames from storage, and a compression library was required for self-extracting code. Implementing this functionality from scratch in assembly language would have been unfeasible.

Our contributions are the following:

First, a *working toolset* which assumes that we have C programs for rendering the data formats to be preserved. Our compiler, based on GCC, supports the full C language, but not all libraries since the targeted abstract machine is limited. We have produced several implementations of the abstract machine using different programming languages: An implementation in F# was developed while still experimenting with different machine features. Later, a C implementation was added with performance as a key goal.

Second, *three complete and alternative descriptions* of the abstract machine, aimed at different audiences:

S1. A guide to building the machine through a series of steps, including tests to check correctness and guide programming. This description should be easy to understand and follow for future programmers with no knowledge of today's hardware, software platforms or tools.

S2. An instruction set architecture (ISA) specification of the machine suitable for contemporary computer scientists.

S3. A self-contained mathematical specification that should be easy to understand for readers with some background in formal mathematics or computer science both today and in the future.

Producing multiple descriptions has helped decide and clarify the features of the abstract machine and avoid ambiguities. Moreover, having descriptions of the machine from two very different angles should be helpful to future readers with varying knowledge and backgrounds.

Third, we have *validated our approach* in two ways:

- We performed an experiment where an inexperienced programmer, unfamiliar with our abstract machine, used the guide (S1) to write a partial implementation; the experiment is described in Section VI-A. This showed that the guide is understandable and that the resources required to produce a partial implementation today are not prohibitive.
- We have a program for testing that implementations of the machine conform to the specification, see Section VI-B. All our implementations passed these tests, including the implementation from the experiment.

Our approach can be realized using any long-term storage technology that can store both digital data and human-readable instructions. Switching storage medium would, however, require new driver code for the storage device.

II. MACHINE DESCRIPTIONS

We have made three different descriptions of the abstract machine aimed at three different audiences, two of which can also be future audiences. In the following sections we illustrate each description by showing fragments of it that involves a running example: `JZ_FWD`, one of two conditional jump

TABLE 1. **The instruction set architecture specification of the `JZ_FWD` operation.**

Hex	Mnemonic	Comment	Immediate	Pop	Explicit effects	Push
03	JZ_FWD	Jump forward on zero	$(1)d$	x	$PC := PC + \mathrm{if}(x = 0, d, 0)$	–

The instruction cycle proceeds as follows:

1) Execute $c := \mathrm{fetch}(1)$, and locate the table entry whose "Hex" column value is c.
2) Execute $x := \mathrm{fetch}(n)$ for every variable, $(n)x$, in the "Immediate" column of the entry.
3) Execute $v := \mathrm{pop}()$ for every variable v listed in the "Pop" column of the entry.
4) Execute all operations listed in the "Explicit effects" column of the entry.
5) Execute $\mathrm{push}(e)$ for every expression e listed in the "Push" column of the entry.

This cycle is repeated until $T = 1$.

instructions in the machine. For reference, the description S2 in reproduced in full in Appendix A.

A. GUIDE TO BUILDING THE MACHINE (S1)

The guide to building the machine S1 is aimed at future programmers, and therefore we assume no knowledge about hardware and software. Also, we use a generic vocabulary and carefully define all terms.

The semantics of each machine operation is described as a series of simple operations that the programmer must implement. To illustrate the guide, here is shown the main loop with only the description of the `JZ_FWD` operation (opcode 03_{16}).[2]

Every time the machine starts, initialize the memory as described in Section 2.6. Then execute the main procedure repeatedly until the value of T is 1_2. The main procedure must carry out the following steps.

1 Initialize the temporary 64-bit element k to 0.
2 Fetch 1 octet into k.
3 If the value of k is

  03_{16} then

    1 Initialize the temporary 64-bit elements a and x to 0.
    2 Fetch 1 octet into a.
    3 Pop x.
    4 If the value of x is 0, then increment PC by a.

4 If the value of k does not occur in the list in the previous point, then set T to 1_2.

The words Fetch and Pop refer to procedures that have been previously defined in a similar manner in the guide.

The guide divides the description of the operations into a set of manageable chunks, each adding another set of operations to the machine. For each chunk there is a small test case, consisting of a program to be entered in the memory and the correct contents of the memory after running it. To facilitate the definition of these test cases, a Common Lisp implementation of the machine was developed in parallel with the description.

B. INSTRUCTION SET ARCHITECTURE (S2)

The instruction set architecture specification S2 is aimed at contemporary computer scientists, including embedded developers, that is, developers writing programs to be run directly on a processor. The semantics of each operation is described as a series of effects in a compact table format. As an example, Table 1 shows only the `JZ_FWD` operation: The operations fetch(), pop(), and push() are described in pseudocode in the complete specification, which appears in Appendix A.

C. MATHEMATICAL DESCRIPTION (S3)

This description, using formal logic, is aimed at future readers with some theoretical background. In order to avoid ambiguities, the description is formulated in a subset of the language of the Coq Proof Assistant [1] and the Coq Equations plugin [2], but no knowledge of Coq or similar systems is required. The description contains some explanatory text, but leaves out the detailed formal proofs. In some cases we also simplified the definitions (compared to the Coq code) since the purpose is to give a precise description of the machine for human readers.

The description is mostly self-contained. In particular, we explain binary representations, monads and monad transformers before defining the machine using a form of "big-step semantics" consisting of:

1) a monad representing the possible states and state transitions,
2) an "implementation" in this monad of a single execution step of the abstract machine,
3) how to construct the machine's initial state,
4) a partial relation expressing the relationship between initial and terminal states.

The core of the implementation (2) starts as follows:

```
Definition exec' opcode : Comp 𝟙 :=
  match opcode with
  | NOP ⇒ return' •
  | JUMP ⇒ pop' ≫= setPC'
  | JZ_FWD ⇒
      offset ::= next' 1;
      x ::= pop';
      if x =? 0
      then pc ::= get' PC;
           setPC' (pc + offset)
      else return' •
  | JZ_BACK ⇒
```

Here $x ::= u;\ v$ is an abbreviation for $u \gg= (\lambda\, x \Rightarrow v)$.

[2]In the guide all non-decimal numbers are explicitly indicated by subscripts indicating the number base in decimal.

III. DESIGN AND IMPLEMENTATIONS OF THE MACHINE

A. DESIGN GOALS

We have designed the machine to be:

1) as simple as possible,
2) a suitable target for a C compiler,
3) reasonably efficient.

By "simple" we mean a machine which is easy to describe, understand and implement. This will not only reduce the implementation effort, but also reduce the potential for misunderstandings. A simpler machine may be less efficient once implemented, which is a disadvantage. However, we assume this to be an inconvenience rather than a problem for two reasons: When some stored data has been presented on a future output device, it can be stored in the output device format so that the processing never has to be repeated. Secondly, if future programmers want to optimize for performance, they can use the simple machine and its code as a starting point.

The machine should be a compilation target for C so that we can use (existing and new) C programs for processing the content formats and encodings.

B. DESIGN CHOICES

1) STACK AND MEMORY

We decided that the abstract machine should be a stack machine, with a program counter (PC) and a stack pointer (SP), but no other registers. The presence of a stack is assumed by C compilers, but we avoid explanations of registers and special instructions for dealing with them.

The machine does not store code and data in distinct parts of memory. With this arrangement, the machine can load code from the storage medium. For example, it can load a format decoder, and then starting using the decoder on data from same medium, all directly using the same execution machinery (process). With a separation between code and data, doing this would have required special instructions in the machine, thus making it more complicated.

2) ADDRESSING

The machine has a 64-bit address space since 32 bits may be too little for some applications. In practice, however, using more than 2^{32} bytes of memory will be slow. Each byte in memory is addressable since this is needed for realizing C byte arrays directly and efficiently. The following instructions have immediate operands: those that push constants onto the stack and (for efficiency) the two branching instructions. The branching instructions also use PC-relative addressing, whereas all other instructions pop addresses and other arguments from the stack. For simplicity, the stack operations all use 64-bit unsigned integers. C has 64-bit integers, and they are efficient on present-day architectures.

3) INSTRUCTION SET

The instruction set of the machine is made up of the following, here organized by purpose: termination and no operation (`EXIT`, `NOP`); jump and branching (`JUMP`, `JZ_FWD`, `JZ_BACK`); setting the stack pointer or pushing the program counter or the stack pointer onto the stack (`SET_SP`, `GET_PC`, `GET_SP`); pushing the constant 0 or immediate operands (`PUSH0`, `PUSH1`, `PUSH2`, `PUSH4`, `PUSH8`); reading and writing to memory (`LOAD1`, `LOAD2`, `LOAD4`, `LOAD8`, `STORE1`, `STORE2`, `STORE4`, `STORE8`); integer or binary arithmetic (`ADD`, `MULT`, `DIV`, `REM`, `POW2`); logical operators (`LT`, `AND`, `OR`, `NOT`, `XOR`); input and output (`PUT_BYTE`, `PUT_CHAR`, `ADD_SAMPLE`, `SET_PIXEL`, `NEW_FRAME`, `READ_PIXEL`, `READ_FRAME`). A detailed description of the instructions is found in Appendix A-F.

4) INPUT AND OUTPUT DEVICES

The machine has a single input device which is used for reading monochrome images from the storage medium. These images are presented to the machine as a set of two-dimensional byte arrays, called *frames*. Each frame can have different dimensions. The encoding used in these frames is handled by the initial program which is included with the machine descriptions, see Section V-A.

The machine has four output devices. The image output device produces two-dimensional arrays of RGB values. There is also a stereo audio device, which can be synchronized with the image output device to simulate a series of images with sound.[3] Finally, there are output devices for Unicode text and raw bytes. We opted for dedicated I/O instructions rather than memory-mapped I/O since explaining the former seemed simpler. The machine's devices are detailed in Section A-E.

5) OMISSIONS

The machine has no native instructions for signed integers. Instead, they are handled using "pseudo-instructions" that are replaced by sequences of native instructions by the assembler. Floating-point numbers are also not part assembly language. Instead, they are handled by the compiler using a C library. Similarly, the machine also leaves all memory management to the software.

6) INSTRUCTION SELECTION

We arrived at this design through an iterative process. Early on, more radical solutions were ruled out since producing an efficient C compiler for these architectures would be very difficult; and floating point numbers were dropped in order to simplify the machine descriptions. Instructions for signed integers were eliminated after benchmarks involving PDF rendering indicated that this had little effect on the performance.[4] In the process, a branching instruction taking a signed byte as immediate operand was replaced by the two separate instructions we have now. This simplified the description while simultaneously improving the performance.

[3]This is not relevant for the currently supported formats.

[4]The binary size increased, but the effect on the execution time was minimal (for the C implementation of the machine mentioned below).

Our instruction set is suboptimal in the sense that most 8-bit values are not opcodes. Adding more instructions could make the binaries more compact and efficient, but we have instead prioritized keeping the machine descriptions short and simple. Since they are easy to explain, instructions have been included for most unsigned arithmetic operators in C, but not for shifting bits to the left or right. This instead has to be implemented using multiplication and division (and a special instruction for computing powers of two). The performance impact of this decision has not been assessed.

7) OTHER ABSTRACT MACHINES AND INSTRUCTION SET ARCHITECTURES

Separately, the features and design choices of our abstract machine are not very original. So why not use an established machine architecture instead? Most importantly, we wanted to try out different choices in order to strike the right balance between the design goals. Moreover, none of the existing machine architectures we looked at turned out to be a suitable starting point. Those having C compilers that produce reasonably efficient programs were too complex to describe and implement, whereas making such a compiler for the simplest architectures would be very difficult. In particular, it is hard to compile C code to efficient programs for machines that do not essentially follow the von Neumann architecture.

Writing a compiler from scratch is difficult regardless of the architecture. Hence, we looked for ways to build on existing work and ended up defining a new target for GCC. Another option we considered was to use as starting point a compiler for WebAssembly [3], adding a post-processing step that translates such bytecode to the simpler instruction set of our machine. However, this would still be a lot of work even if we were to choose a subset of WebAssembly as our instruction set. The fact that WebAssembly is safer and (in some sense) higher level than C also create some complications. In particular, it would be difficult to include I/O instructions as inline assembly code.

C. CURRENT IMPLEMENTATIONS

The main idea of our approach is that people in the future should be able to consume our data (even if the data formats are no longer understood) by implementing the abstract machine using the tools at their disposal. Nevertheless, we have also implemented the machine ourselves.[5]

1) IMPLEMENTATION IN F#

We first implemented the machine in F# and used this implementation to experiment with different architectures and instruction sets. Performance was never a priority for this implementation, but simplicity and conceptual clarity. Thus, it looks very much like the mathematical description in Section II-C:

```
member m.Step () =
  match m.NextOp 1 |> int8 with
  \ldots
  | JUMP -> m.PC <- m.Pop ()
  | JZ_FWD ->
    let offset = m.NextOp 1
    if m.Pop () = 0UL
    then m.PC <- m.PC + offset
  \ldots
```

The F# implementation also has some rudimentary features for debugging.

2) IMPLEMENTATION IN C

When we started to compile more complex programs, there was a need for a faster implementation, which we wrote in C. Like the F# version, it is just a simple interpreter which executes the instructions one by one:

```
 switch (next1()) {
   case EXIT: goto terminated;
   case NOP: break;
   case JUMP: pc = (void*) pop(); break;
   case JZ_FWD:
     x = next1();
     if (pop() == 0) {
       pc += x;
     }
     break;
   ...
```

Nevertheless, this implementation is relatively fast, partially at the expense of memory safety: Since the assembler produces programs that are position independent, we can represent memory addresses using C pointers.[6]

D. THE ASSEMBLER

Since the work on the compiler had to start before the instruction set was finalized, we decided to define a distinct assembly language as an abstraction barrier. The corresponding assembler also handles issues such as the fact that the native branching instructions can jump at most 256 addresses. Rather than defining a new target architecture for an existing assembler, we implemented the assembler from scratch in F# using the FParsec parser combinator library [4].

1) ASSEMBLY LANGUAGE

The assembly syntax we use is reminiscent of the GNU Assembler, but with some important differences. The complete grammar is included in Appendix B. The assembly language has many more instructions than the abstract machine. Additional "pseudo-instructions" are converted to sequences of machine instructions by the assembler. Notable pseudo-instructions are those that deal with signed integers and the unlimited branching instructions. Since the number of machine instructions needed for branching depends on the

[5]In addition to the implementations mentioned here, a second C implementation was developed while working on the compiler, and we have partial implementations in Python and Common Lisp.

[6]This assumes that the computer running the implementation has 64-bit addresses and little-endian memory representation like our abstract machine.

jump distance, the assembler needs several passes to get the references right. Some optimizations have the same effect. For instance, a short jump may be translated into a PUSH0 followed by JZ_FWD or JZ_BACK.

The assembly language has several features intended to make writing programs for a stack machine less confusing. They are particularly helpful when writing assembly programs by hand. The following recursive subroutine outputs a 64-bit unsigned integer in decimal notation:

```
write_decimal_number:
  ## 0: return address, 1: argument
  push! (/u $1 10)
  jump_zero!! $0 next
  call! write_decimal_number  # Recursion
next:
  set_sp! &1
  zero = 48
  put_char! (+ (%u $1 10) zero)
  return
```

Observe that the instructions can take complex expressions (in prefix notation) as arguments. The assembler translates these into stack operations. Here &*n* and $*n* denote the address and value of the *n*'th (64-bit) element on the stack before the statement is executed; and *m* consecutive exclamation marks following an instruction indicate that the values of *m* expressions should be pushed to the stack first. Otherwise, the instruction must get its arguments from the existing stack. For instance, `return` is an alias for `jump`, which means "jump to an address popped from the stack".

2) ASSEMBLER IMPLEMENTATION

Here is some of the code involved in the code generation for the `jump_zero` assembly instruction:

```
let bDist x = -256L<=x && x<=255L

let jz (d: int64) =
  if d >= 0L
  then [JZ_FWD; int8 d]
  else [JZ_BACK; int8 <| abs d -1L]

let deltaJump d =
  if d = 0L then []
  elif bDist <| d - 3L then [PUSH0] @ jz (d - 3L)
  elif d<0L then [GET_PC] @ pushNum (d-1L) @ [ADD;JUMP]
  else
    let f pushLength = d - int64 pushLength - 1L
    pushFix f @ [GET_PC; ADD; JUMP]

let deltaJumpZero d =
  if bDist <| d - 2L then jz (d - 2L)
  else
    let jump = deltaJump (d - 5L)
    jz 3L @ [PUSH0] @ jz (opLen jump |> int64) @ jump
```

The assembler generates position independent binaries, but tries to simplify expressions before generating the machine instructions. For instance, `(+ label1 -label2)` becomes a constant. When necessary, the generated machine code is prefixed with code that (at runtime) adjusts the contents of data blocks referring to labels.

The assembler output is the program binary, as it should be loaded into the memory of the machine, and a text file containing the relative locations of labels that can be used for debugging and a limited form of linking. However, it is not possible to link these files like traditional object files. Currently, the compiler gets around this by letting assembly files play the role of object files. The linker essentially concatenates these files before calling the actual assembler, see Section IV for details. It is also possible to leave more of this to the assembler, which supports explicit import statements but also leaving these implicit.

The assembler comes bundled with the F# implementation of the abstract machine and has support for running the generated binary without first having to write it to disk. It also contains a basic testing framework which assembles and runs a program before checking the final stack. This has been very useful for catching bugs and avoiding regressions.

IV. THE COMPILER

The role of the compiler is to provide an assembly representation of the C programs, that eventually will be converted to binary by the assembler (Section III-D). The assembly language serves as an interface between the compiler and the assembler. Actually, the developed compiler is a C cross-compiler that must be compiled and executed on today's platforms (such as x86-64).

A. BASE COMPILER INFRASTRUCTURE

One of the key points in the design of the C compiler is the adoption of an open-source compiler infrastructure as the base system. We adopted GCC (GNU Compiler Collection) for the compiler for several reasons: it is widely used, it has a very stable design API between different versions, and it is a full C compiler.

The core of the GCC C compiler (called `cc1`) is in charge of translating the C language into the *target* assembly language. Fig. 1 sketches the structure of `cc1`. Two intermediate representations are used in different stages of the compilation: the architecture-independent GIMPLE representation for the abstract syntax tree passes, and the RTL (Register Transfer Language) representation for those passes that are target dependent. The translation of the GIMPLE format into RTL is known as *expansion*. The expansion gives rise to a sequence of RTL expressions (RTX) that can be seen as a coarse approach of what will be the target assembly code. An RTX is a sort of a Lisp expression.

The RTL makes use of the concept of *register*. The expansion pass considers an infinite number of registers when the first RTL is emitted, prior to the rest of transformations. These symbolic registers are called *pseudoregisters*. Pseudoregisters will finally be mapped to actual (architectural) registers of the target architecture. This operation is known as *register allocation*, which is carried out by the *reload* pass.

During the *final* pass, the RTX sequence is converted into assembly statements. Designing a target involves providing a *machine description* that specifies the features of

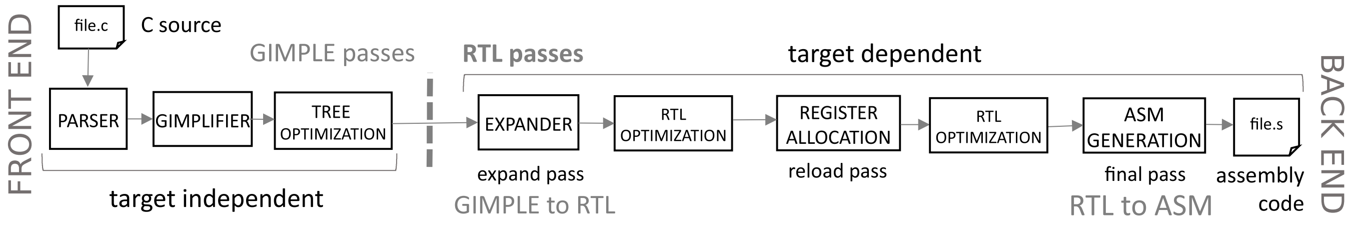
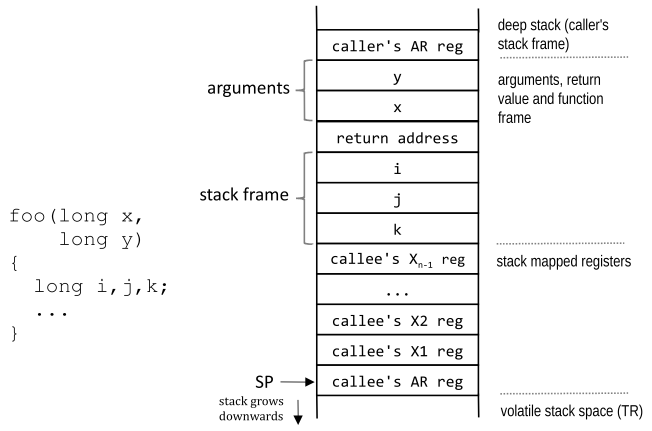


FIGURE 1. Overview of `cc1` stages (some boxes like optimizations may involve many compiler passes).

#	Name	Meaning
0	SP	Stack Pointer
1	FP	Frame Pointer
2	TR	Top of Stack Register
3	AR	First general purpose register / Return value
4-18	X1, ... X15	Other general purpose registers

FIGURE 2. Registers defined for compilation purposes.



```
foo(long x,
    long y)
{
  long i,j,k;
  ...
}
```

FIGURE 3. Example of the stack layout after the prologue of a given function.

the architecture. The GCC machine description defines valid RTL patterns, fulfilling certain constrains such as the addressing modes, as well as which assembly instructions will be generated.

B. STACKIFICATION SCHEME

One of the main challenges in designing the GCC backend for the architecture of the abstract machine has been the lack of registers. There are neither general-purpose registers nor a frame pointer (FP). This fact requires reworking the *register allocation* phase. An approach similar to that in [5] has been followed. At first, a set of *target* general-purpose registers is assumed. These registers are used as architectural registers during the register allocation phase, but they are finally mapped to stack positions when the assembly code is generated.

Fig. 2 shows these *target* registers. The PC is an implicit register. The FP register is used by the compiler until the register allocation phase, where it is removed (as it is not an actual register in the abstract machine). Basically, FP is expressed as a function of SP, according to certain elimination rules set by the target description.

TR is an instrumental register that represents the top of the stack (TOS). Writing to it involves pushing a word on the stack. Reading from it involves popping a word from the stack. This is not a general-purpose register, but it is used to express temporary computations requiring moving data from/to the stack, and consequently it must only be handled following certain rules.

Finally, 16 general-purpose registers are defined (this number was decided experimentally), named as AR, X1, X2, …X15. The AR register is used by functions to return a 64-bit value (or smaller).

Fig. 3 shows the stack layout allocated immediately after the call to a given function, including the stack-mapped registers. Observe that the target must track SP in order to locate the current position of the first general-purpose register. In a function, SP can change in two ways: (1) explicitly, for example, when pushing an argument on the stack (SP is pre-decremented), and (2) implicitly, for example, when operating on TR (in that case the GCC machinery has no knowledge about this SP modification).

C. INSTRUCTION EXPANSION

During the *expansion* pass (see Fig. 1), the internal tree representation (GIMPLE) is expanded via a set of canonical RTL rules that must be included in the machine description file. It is during this phase that the instrumental register TR is used to express those temporary operations performed on the top of the stack.

Observe that TR is a special register, and GCC has to be instructed not to use it in a general way. One must avoid performing transformations that give rise to wrong programs, for example, due to a stack imbalance. With this aim, some operations on TR have been defined in the machine description using the `unspec` RTL clause, which imposes certain restrictions on the compiler when transforming these RTL expressions. For example, consider the transfer `dst` $\leftarrow$ `TR - TR`. What does it mean? Perhaps its meaning is not `dst` $\leftarrow$ `0`, as probably the GCC machinery would infer. Perhaps the developer's purpose is to express popping two operands from the stack, subtracting them and writing the result to `dst`. The TR register should be used only for operations and transformations allowed in the machine description, but for nothing else. These RTL patterns define basic actions like

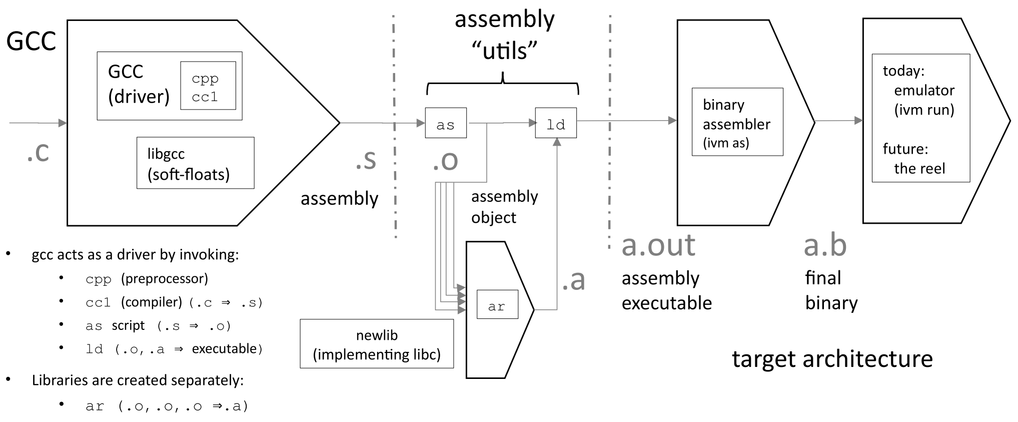


FIGURE 4. GCC toolchain for the abstract machine target.

push (TR ← operand), pop (dst ← TR) and arithmetic operations (TR ← TR op operand), among others.

An example illustrating the expansion process is included in Appendix C-A.

D. COMPILER TOOLCHAIN

One of the goals when designing the C compiler has been to keep things as conventional as possible, in such a way that existing C projects could be ported and built with minimum effort (for example, consider the effort of adapting configuration scripts, makefiles, etc.).

In the standard compilation flow, gcc acts as a *driver* that invokes several other programs: the C preprocessor (cpp), the compiler itself (cc1), the assembler (as) and the linker (ld). First, cpp translates the input C source file (for example, prog.c) into an ASCII intermediate file (prog.i), which cc1 translates into an ASCII assembly language file (prog.s). Second, the assembler translates the assembly file (prog.s) into a relocatable object file (prog.o). Third, the linker combines one or several object files (including those in libraries) into a single executable file.

No linkable binary object format was defined. As a result, the above standard compilation flow needed to be adapted. This led to the development of a compiler toolchain that works completely in assembly format, deferring the binary generation to external tools, after the linking process. With this purpose, the following file types were defined:

- *Assembly Object*: The result of applying the program as to the cc1 output. The extension .o is used for these objects. Basically, it is the same assembly file generated by cc1 where a unique suffix is added to the local symbols (local symbols refer to labels or abbreviations that are not declared as global with the EXPORT clause). The aim of this renaming process is to assure that local symbols in one object will not conflict with other local symbols (with the same name) in another object when combining several objects together during the linking process.
- *Assembly Executable*: The result of combining multiple *assembly objects*, including those coming from libraries. This linking process is carried out by the ld program.

Fig. 4 shows the compilation flow. Programs as and ld have been implemented as shell scripts. Note that the as script shown here is not the assembler tool in charge of generating the final binary.

The ld script has the ability to create a true executable by means of a *shebang* header added to the final assembly output. This feature is especially useful for testing the compiler output on today's platforms, where the compiler has been compiled. This way, the final output of the compiler, the *assembly executable*, is a concatenation of: (1) a shebang header, that enables its execution; (2) the crt0.o startup file, which is placed at the very beginning; (3) all object files of sources being compiled (which are really assembly files); and (4) the standard C libraries, and other libraries provided on the command line.

E. REMARKS ON THE COMPILER DEVELOPMENT

The compiler backend for the abstract machine has been designed for GCC version 10.2.0 which has resulted in a robust and efficient C cross-compiler. It should be mentioned that the development of the backend has involved a great effort of validation and experimental work, not only with the specific applications but with many other benchmarks (see Appendix C-E). To carry out an efficient testing, some companion tools have been developed, such as a fast abstract machine emulator and an in-RAM file system generator. As a final remark, we are considering extending the compilation system for C++, in order to support other format renderers written in that language.

V. PROGRAMMING THE MACHINE AND ITS LIBRARIES

The software for the abstract machine is primarily compiled from C, but with some parts in assembly language such as in the I/O library, which must use the native I/O instructions. We also use some assembly language in the initial program to keep its size down.

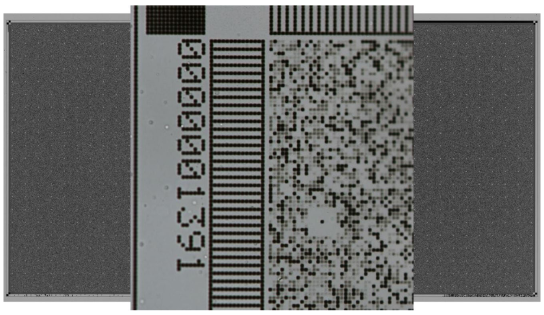


FIGURE 5. A film frame in boxing format, with a magnification of the upper-left corner superimposed.

A. THE INITIAL PROGRAM
Most of the code that is executed by the machine is read from the digital storage medium via the input device. To start the process of reading from the input device, an initial program must be in place. This initial program depends on the concrete storage medium in use; thus the following discussion will refer to "frames" and the "boxing library" (Section V-C) that decodes frames as data.

The initial program uses a fragment of the boxing library to load rendering code and the full boxing library from the film, but cannot be loaded like this itself. Instead, it must be printed in human-readable form on the storage medium so that it can be placed in the memory of the machine before it is started. Self-extracting code is used to make the initial program as short as possible. We use a specially adapted version of the XZ Embedded compression library intended for embedded systems where small code size is important. To further reduce the size of the compiled code, all libc calls have been replaced with as trivial equivalents as possible.

B. INPUT AND OUTPUT
To add support for preserving a new file format, one needs a C library for the format. Such libraries are available for most image and document file formats used in digital preservation. The library has to support decoding from memory buffers instead of relying on file I/O. Many support this out of the box, either through the library API or compile time settings. The library also cannot use files for temporary storage of data, and it must be adapted to use the I/O library of the abstract machine. This is a simple C library that lets programs access the input and output devices.

C. DECODING INPUTS: THE BOXING LIBRARY
On the film storage medium of Section VII-C, digital data is stored in high capacity 2D-barcodes written to the film as monochrome images, see Fig. 5. Since most commonly available 2D-barcode formats have relatively low capacity, the company selling the film has developed a custom format for it. This format is called the boxing format, and is supported by an open-source library written in C.

D. SUPPORTED FORMATS
1) PDF/A
The PDF renderer is based on Ghostscript version 9.52. The core of Ghostscript is a Postscript interpreter implemented in C, and the PDF rendering program is implemented as Postscript code that is executed on this interpreter.

The Ghostscript C code is portable, so it can be built on many different machine architectures and operating systems. However, to make the code build and run successfully on the machine, it was necessary to make some modifications to it. We also had to write a simple device driver to interface the program with the machine's graphical output device.

2) JPEG AND TIFF
The TIFF renderer is libtiff version 4.1.0 and the JPEG renderer is version 9d of 12-Jan-2020 from Independent JPEG Group. Both are mature and flexible C implementations that have been deployed to many hardware architectures and operating environments, often with limited hardware resources. Therefore, supporting preserving these data formats required very little effort.

VI. VALIDATING THE GUIDE TO BUILDING THE MACHINE
The guide to building the machine (S1) is essential to the success of our approach, and thus we have validated it. This took the form of an independent implementation of the guide by a programmer not familiar with the machine descriptions or with the general work.

A. EXPERIMENT
To validate the guide to building the machine we had a programmer implement a machine using the guide. The programmer was given a previous version of the guide and nothing else.

a: THE PROGRAMMER
The programmer is a researcher at the Norwegian Computing Center with a master's degree in image processing and computer vision, and a PhD in image processing. He had previously worked 2 years as a software developer. On starting his task he had only superficial knowledge of the work described in this article, but he was told that the purpose of the machine was long-term storage.

b: TASK EXECUTION
He spent 42.5 hours implementing a machine while keeping a brief development log and submitting code to a version control repository. The time spent included 3 hours documenting his results and commenting in writing on the guide. The work was done over a one-month period, interspersed with other unrelated tasks. He did not consult with others on his task except for the following: He got confirmation on two typographic errors that made tests given in the guide fail unintentionally, and he got explained the mathematical part of the definition of the modulo operation.

c: SCOPE
During the experiment we decided to reduce the scope of the task by leaving out the implementation of the input and output device support of the machine. The reasons for this choice were that programming device support is time-consuming since it depends on the implementation platform's own I/O support and furthermore that testing I/O support is difficult. (The guide presently does not have tests for device handling.)

d: RESULTS
We left the choice of programming language to the programmer and he chose Python. He was able to implement the whole machine without I/O devices and his code passed all tests in the guide and the test program (Section VI-B).

e: REVISIONS TO THE GUIDE FOLLOWING THE EXPERIMENT
After the experiment the guide was revised. The description of the memory elements was improved to clarify the structure of the memory, and a figure was also added to show this structure graphically. Several details in definitions and tests were also clarified. Comments were added to the mathematical description of the integer division and remainder operations, to clarify informally the purpose of these.

The experiment revealed two errors that were corrected: In one test an incorrect variable was given, and in another test a constant was wrong.

B. TEST PROGRAM
The test program for the abstract machine is an assembly program that tests every branch of every instruction at least once, thus providing full path coverage. Care was also taken to detect errors that could potentially arise due to certain types of corner cases in the operands. An important class of corner case that was handled was detection of incorrect implementation of instructions through the use of signed arithmetic instead of unsigned.

The following tests were designed to detect corner cases in the operands:

- The two conditional branch instructions are tested both when the condition is true and when it is false.
- The "less than" test is tested on four different combinations of numbers, testing for when the operands are equal, greater than, and less than, and also includes a case when one operand has the most significant bit (MSB) set (which would give an incorrect result if signed arithmetic were used by the implementation).
- All two-argument arithmetic and logical operations are tested on two pairs of operands, both of which include one number that has the MSB set.
- The bitwise "not" operation is tested on two different operands.
- The binary power operation is tested on three different operands.

C. DISCUSSION
During the experiment, the programmer ran tests described in the guide itself. After the experiment was over, the experimental implementation was tested with the test program (Section VI-B), and it passed all tests.

VII. PRESERVATION, RETRIEVAL, AND STORAGE
Here we explain how to preserve and retrieve data using our approach: today using the toolset, and in the future building a machine and using it to read and present the data from the storage medium. We also discuss the general archival process and a concrete storage medium that can be used.

A. PRESERVING AND RETRIEVING INFORMATION
1) PRESERVING INFORMATION WITH THE TOOLSET TODAY
To enable the toolset to support a new data format, X, one must do the following:

1) Acquire a C language library, L_X, for decoding and rendering data on format X.
2) Adapt L_X if needed to support reading from memory buffers and decoding to memory buffers.
3) Write C code that feeds the L_X decoder with the file memory buffers read from the storage device.
4) Write C code that reads the decoded memory buffers and converts it to a format supported by the machine output devices.
5) Use the C compiler to compile an updated version of the toolset.
6) Update the table-of-content metadata to indicate that the file format is supported.

Next, to preserve data one must do the following:

1) Write human-readable images of the descriptions to the analog storage medium: the guide to building a machine, S1, the mathematical description, S3, and the initial program.
2) Encode both the archival and rendering software using the boxing library.
3) Write the encoded data to the digital storage medium.
4) Store the storage media in a suitable location.

2) RETRIEVING INFORMATION IN THE FUTURE
Once future historians and their helpers have discovered the storage location, they should do the following:

1) Use one of the descriptions to implement the abstract machine. (Presumably, it will be most efficient to digitize the storage medium in advance and let input instructions read from these files. Perhaps the output devices should write to files as well, instead of trying to present the data directly to the user.)
2) Then, either by hand or using an optical character recognition device, enter the initial program (see Section V-A).
3) Start the machine with the initial program in memory.

This will render the contents of the digital storage medium to the future physical output devices (or extract the original files if the start configuration of the machine is adjusted).

B. THE ARCHIVAL STORAGE PROCESS

The storage process is typically performed as a subprocess of maintaining a digital preservation system following the guidelines in the OAIS process.[7] This framework recommends creating an archival information package (AIP) that is stored in the digital archive. The AIP is typically produced by a specialized archiving software, for example Archivematica, that produces an AIP with the original data, fixity information and metadata. In this process the AIP is added to the archival file system (AFS) that is written to the storage medium. The AFS has some unique features that is not common for file systems, such as the ability to reference both analog and digital versions of the same file from the file system index (table of contents). The AFS is agnostic in terms of physical storage medium, but a requirement is that it can hold analogue 2D images with at least two density levels and can resolve symbols at MTF 70% or more. The minimum storage size of the AFS is called a frame, corresponding to a sector on a hard drive. The frame has a border with optical recognition tracking codes and human readable frame index. Inside the frame is the payload with digital data or analog images, where the digital data is protected by intra and extra frame forward error correction. The sequence of frames is called a reel, and the first frame in the reel is called the "control frame". The control frame can be compared to the master boot record of hard drives, as it contains information and indexes to the rest of the content on the reel, including the table of contents.

C. PHYSICAL STORAGE MEDIUM

Physical storage of both visual instructions and digital data can be done using a special film made by the company Piql,[8] This film is certified using ISO-based accelerated lifetime test procedures for 1000 year of storage. Film has several advantages: It lasts for a long time if stored properly; it can store both analog information, that is, visual information, and digital information on the same medium; and the visual information can be read with a one-lens element magnifying glass. The disadvantage is that the digital information must be decoded from an analog representation on the film. Storing information more directly and efficiently seems to require some kind of hardware, and hardware has a limited lifetime. One strategy to counter the hardware lifetime problem is to repeatedly migrate the data to a new technology when the current hardware technology nears its end of life. Note, however, that in this case the problem of software lifetime remains.

Piql stores films for its customers in the Arctic World Archive,[9] inside a decommissioned coal mine near Longyearbyen at Svalbard. The archive is in use by archival institutions, museums and other organizations from more than 15 countries, including the Vatican Library, the European Space Agency, and GitHub. The latter has deposited 21 TB of their open-source software repositories.

The current write speed for this storage medium is 40 MiB/s, and the goal is to support that speed or higher in all parts of the creation processes. This is not affected by our approach since the digital data is stored in its original format. Writing the machine descriptions and software to each reel incurs a small constant overhead, but compilation can be done once in advance.

VIII. RELATED WORK

The idea to essentially use machine code as a universal data format is related to that of the universal Turing machine, but it was first suggested as an approach to long term preservation by Lorie [6], which introduced the notion of a Universal Virtual Computer (UVC). In subsequent work a concrete instruction set architecture (ISA) was suggested, but to our knowledge no compiler has yet been created. Their ISA is also more complex and thus harder to implement than our abstract machine, and it does not specify concrete output devices. Instead, their programs will produce a "logical view" of the preserved documents, for example, as XML.

Nguyen and Kay [7] have a simple abstract machine, and their stated goal is that it should be possible to implement this machine in an afternoon following its one-page description. They also demonstrated SmallTalk-72 running in an implementation of this machine, but there is no compiler, and their focus is more on preserving interactive software than digital information. Joguin [8] aims for preserving software on film much like us, but they too seem to miss a compiler producing reasonably efficient code. Appuswamy and Joguin [9] aim for database preservation. Braun *et al.* [10] have considered secure long-term storage, but our machine has no security features since they would only complicate it; compare for example with WebAssembly [3].

What makes our approach unique is primarily the fact that we have a fully working solution that handles complex formats, and that there is a straightforward way to add support for more such formats. Among other things, this means that we have a C compiler, concrete descriptions of the abstract machine for future readers, and a self-extracting initial program to "bootstrap" the retrieval process. We also believe that our machine strikes a better balance between simplicity and efficiency than the alternatives.

We now discuss work related to the compiler. A good starting point when porting GCC to a new architecture is analyzing existing targets, since GCC has been ported to an ample number of architectures, either brand new or old ones. The main reference on porting GCC to a new target is [11]. Some guidelines can, however, be found elsewhere [12], [13]. Focusing on pure stack machines, the number of targets is scarce. We can highlight two of them: the Thor [5] and ZPU [14] processors.

The Thor processor is a 32-bit CPU developed for aerospatial applications. It is a relatively old project (gcc 2.7, 1995), where a GCC backend for this target was developed. The

[7] http://www.oais.info/standards-process/
[8] https://www.piql.com
[9] https://arcticworldarchive.org

TABLE 2. Sizes of the artifacts of our approach to long-term preservation.

Artifact	Guide (S1)	ISA (S2)	Math (S3)	F# machine	C machine	Assembler	Compiler+libs	Initial program
Size	17 pages	5 pages	12 pages	390 LOC	532 LOC	2 500 LOC	11 100 LOC	2 685 LOC

processor is a pure stack machine with no registers other than the PC and SP, although the architecture is slightly more complex than that of our abstract machine. This project contributes some valuable ideas of how to handle the lack of registers.

ZPU is another small zero-operand, 32-bit, simplistic architecture. The GCC toolchain sources are available in [14] (gcc 3.4.2, 2004). These sources include the compiler, the assembler, the linker and other helper programs. Unlike Thor, which makes the stackification of operands as soon as possible (in the expansion pass), the ZPU backend postpones it to the final pass.

IX. FINAL REMARKS

The idea to use a machine for long-term preservation is not new. However, our approach distinguishes itself from previous attempts in two key ways: Our simple machine has two alternative future descriptions assuming very different reader backgrounds, and crucially, we have a C compiler that targets the machine. Together this allows to preserve complex formats such as PDF/A, and it significantly reduced our engineering efforts since programming the machine could partly be done using C. Table 2 indicates some of the effort that went into writing the machine descriptions and the code that make up our approach.[10] All of our code is open-source and available on GitHub[11] together with the descriptions of the machine.

A. PERFORMANCE

The cost efficiency of our approach to long-term preservation depends on multiple factors, some of which have been outside the scope of this work, for example, the performance characteristics of the storage medium and its representation/encoding of analog and binary information. Instead, we have focused on:

1) The size in the analog (that is, human-readable) storage medium of the abstract machine descriptions and the initial program.
2) The size in the digital storage medium of the binaries for archival and rendering.
3) The cost and ease of implementing the abstract machine in the future.
4) The cost and ease of porting rendering software for additional formats today.

Observe that all these costs can potentially be amortized across many storage instances.

[10]We also did a lot of work in Coq, see Appendix B.

[11]https://github.com/immortalvm; some parts coming by the end of December 2021.

Regarding item 1: This part takes up 0.07% of the film, computed as follows: The how-to guide and mathematical specification together take up 29 pages, cf. Table 2. We assume that one page of a document takes up a whole frame on film. The initial program is 230 KiB. This will use 14 frames on film when hex-encoded using a font size of 12×20.

Regarding item 2: This part is 13,8 KiB,[12] which takes up 8 frames or 0.12% of the film if we use the film storage medium of Section VII-C (where one film has 65 000 frames).

Regarding the latter two items: These cannot be assessed quantitatively, but we believe that our work demonstrates that the cost today is reasonable; the cost of future implementations is hard to estimate, but our results—in particular the experiment (Section VI-A)—indicate that it should be feasible.

The PDF rendering performance was benchmarked using the C implementation of the abstract machine (III-C2) in a VirtualBox virtual machine running on an Intel Core i7-8850H (2.60GHz). The three test documents were rendered at a resolution of 300 ppi, with 16 levels of anti-aliasing.

	Seconds per page
17-page technical document	30
96-page government report	83
378-page illustrated book	63

B. FUTURE WORK

Even though we have a working solution which is ready for use today, there are some questions that should be investigated further. Most importantly, our approach to long-term preservation should be compared to the alternatives, ideally by an independent research group. We would also like feedback from more developers implementing the machine in order to see if the descriptions can be simplified further or improved in other ways.

Another important question is how to avoid that the code for our abstract machine becomes yet another troublesome format. For instance, the people of the future should not have to implement multiple abstract machines with subtle differences. If our approach catches on, there will inevitably be suggestions to change the machine in various ways, whether for performance reasons or to add more features such as interactivity. Thus, we should at least introduce version numbers for the machine specifications. Ideally, we should also find a way to make new versions backward compatible.

[12]This number was computed by summing the size of different binaries.

One reason why the machine might change is that not all design choices have been obvious or deeply investigated. We did not have a precise way to measure how the alternatives would affect the cost and ease of implementing the machine in the future. Moreover, even measuring the effect of these choices today (on the size and speed of the machine code) was difficult since this ultimately depends on possible compiler optimizations. Having an intermediate assembly language meant that we could experiment with the native instruction set without constantly rewriting the compiler; but it also means that any conclusions—for example, with regards to the effect of having no signed instructions—must be taken with a grain of salt.

Even if we keep the current machine fixed, there are many potential improvements. More optimizations can be added to the compiler, and one might reduce the size of the initial program considerably by initially using a more primitive encoding (that is, before the main program is loaded). Whereas our abstract machine should be easy to implement correctly even in the far future, other components of our system are quite complex and thus likely contain errors. This risk can be reduced by rendering any content to be stored using one of our current machine implementations and comparing it to an authoritative rendering. We hope to add this to our process as an optional step in the future, even though it will be quite slow for some media types. It would also be nice to prove more formal properties, for example, that the assembler is correct.

Finally, we believe our approach has a potential beyond the preservation of media files. There is already a way for the user to pass options and other data to the machine. Thus, it can be used to render contents from other sources than the current storage instance. It should also be possible to compile the compiler itself so that our approach can be used to preserve software artifacts as well.

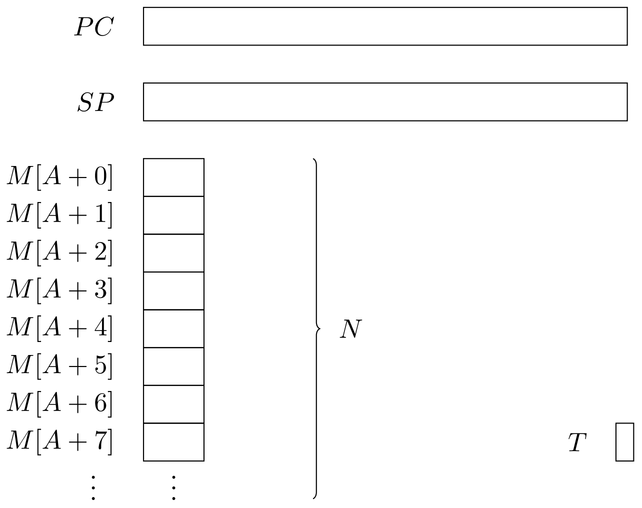
APPENDIX A
INSTRUCTION SET ARCHITECTURE SPECIFICATION

A. PROGRAMMING MODEL

The IVM is a pure stack-based machine: it has a program counter and a stack pointer, but no general-purpose registers. The programming model consists of the following elements:

- *Memory*: An array of 8-bit locations, $M[A..A+N-1]$, where $0 \leq A < 2^{64}$ and $0 < N \leq 2^{64}$.
- *Program Counter*: A 64-bit register, PC, that points to the next instruction to be fetched or to any immediate operands of an instruction. The initial value of PC is A.
- *Stack Pointer*: A 64-bit register, SP, that points to the top of the stack, which is the memory region from $M[SP]$ to $M[M+N-1]$, inclusive. The initial value of SP is $(M+N) \bmod 2^{64}$.
- *Terminate Flag*: A 1-bit flag, T, that is set to 1 when the machine has terminated. The initial value of T is 0.

These elements are shown graphically in the following figure.



B. BASIC DEFINITIONS

Definition 1 (Floor): $\lfloor x \rfloor$ is the unique integer such that $\lfloor x \rfloor \leq x < (\lfloor x \rfloor + 1)$.

Definition 2 (Integer division):

$$x \operatorname{div} y = \left\lfloor \frac{x}{y} \right\rfloor$$

Definition 3 (Modulo):

$$x \bmod y = x - y \left\lfloor \frac{x}{y} \right\rfloor, \quad y > 0$$

Definition 4 (Bit value): For any integer value, x, the notation $x.\text{bit}[i]$ refers to the value of bit i in x.

Definition 5 (Octet value): For any integer value, x, the notation $x.\text{octet}[i]$ refers to the integer made up of the bit sequence from $x.\text{bit}[8i+7]$ to $x.\text{bit}[8i]$, inclusive.

C. BASIC FUNCTIONS

The following functions are needed by some instructions. For each function, its arguments and its result are all 64-bit integer values, except where otherwise noted.

Definition 6 (Conditional):

$$\text{if}(e, c, a) = \begin{cases} c & \text{if } e \text{ is true} \\ a & \text{otherwise} \end{cases}$$

Definition 7 (Addition):

$$\text{add}(x, y) = (x + y) \bmod 2^{64}$$

Definition 8 (Multiplication):

$$\text{mul}(x, y) = (xy) \bmod 2^{64}$$

Definition 9 (Integer division):

$$\text{div}(x, y) = \begin{cases} q \mid x = qy + r \wedge 0 \leq r < y & \text{if } x > 0 \wedge y > 0 \\ 0 & \text{otherwise} \end{cases}$$

Definition 10 (Integer remainder):

$$\text{div}(x, y) = \begin{cases} r \mid x = qy + r \wedge 0 \leq r < y & \text{if } x > 0 \wedge y > 0 \\ 0 & \text{otherwise} \end{cases}$$

Definition 11 (Binary power):

$$\mathrm{pow2}(x) = \begin{cases} 2^x & \text{if } x < 64 \\ 0 & \text{otherwise} \end{cases}$$

Definition 12 (Bitwise boolean "and"):

$$\mathrm{and}(x, y) = z \mid \forall i \in \{0, \ldots, 63\}$$
$$\mid z.\mathrm{bit}[i] = \begin{cases} 1 & \text{if } x.\mathrm{bit}[i] = 1 \wedge y.\mathrm{bit}[i] = 1 \\ 0 & \text{otherwise} \end{cases}$$

Definition 13 (Bitwise boolean "or"):

$$\mathrm{or}(x, y) = z \mid \forall i \in \{0, \ldots, 63\}$$
$$\mid z.\mathrm{bit}[i] = \begin{cases} 1 & \text{if } x.\mathrm{bit}[i] = 1 \vee y.\mathrm{bit}[i] = 1 \\ 0 & \text{otherwise} \end{cases}$$

Definition 14 (Bitwise boolean "not"):

$$\mathrm{not}(x) = z \mid \forall i \in \{0, \ldots, 63\}$$
$$\mid z.\mathrm{bit}[i] = \begin{cases} 1 & \text{if } x.\mathrm{bit}[i] = 0 \\ 0 & \text{otherwise} \end{cases}$$

Definition 15 (Bitwise boolean "exclusive or"):

$$\mathrm{xor}(x, y) = z \mid \forall i \in \{0, \ldots, 63\}$$
$$\mid z.\mathrm{bit}[i] = \begin{cases} 1 & \text{if } x.\mathrm{bit}[i] \neq y.\mathrm{bit}[i] \\ 0 & \text{otherwise} \end{cases}$$

D. MEMORY ACCESS PROCEDURES

Five basic procedures for memory access are used as building blocks for the instructions.

1) PSEUDOCODE ELEMENTS

We introduce the following pseudocode elements to describe the procedures in this section.

- $x := v$
  Assign x the value v.
- **var** $x := v$
  Declare variable x, assigning it the value of v.
- **for** i **in** $m \ldots n$ **do** $S(i)$
  Evaluate $S(i)$ $n - m$ times, with i successively bound to every integer in the range $\{m, \ldots, n - 1\}$.
- **return** v
  Return the value of v.

2) GENERAL MEMORY ACCESS OPERATIONS

There are two basic procedures for general memory access that instructions can use to store integers to or load integers from a given memory address. The memory is 8 bits wide, and integers can be 8, 16, 32, or 64 bits wide, so they are stored from the given memory address in little-endian format. An 8-bit integer is trivially mapped to the specified memory address.

The procedure store(n, a, x) stores an integer, x, as n octets starting at memory address a. It is defined in pseudocode as follows:

$$\mathrm{store}(n,\ a,\ x) \equiv \textbf{for } i \textbf{ in } 0 \ldots n \textbf{ do } M[a+i] := x.\mathrm{octet}[i] \tag{1}$$

The procedure load(n, a) returns an integer loaded from n octets starting at memory address a. It is defined in pseudocode as follows:

```
load($n$, $a$) ≡ var $x := 0$
                 for $i$ in $0 \ldots n$ do $x.\mathrm{octet}[i] := M[a + i]$
                 return $x$                                        (2)
```

3) STACK OPERATIONS

The stack operations are defined in terms of the general memory access operations, using the stack pointer as the memory address. All stack operations work on 8 octets at a time, so arguments and results are assumed to be 64-bit integers. For this reason the stack operations also decrement and increment the stack pointer in multiples of 8.

The procedure push(x) pushes an integer, x, on the stack. It is defined in pseudocode as follows:

```
push($x$) ≡ $SP := (SP - 8) \bmod 2^{64}$
            store(8, $SP$, $x$)                                    (3)
```

The procedure pop() returns an integer popped off the stack. It is defined in pseudocode as follows:

```
pop() ≡ var $x := \mathrm{load}(8,\ SP)$
        $SP := (SP + 8) \bmod 2^{64}$
        return $x$                                                 (4)
```

4) FETCH OPERATION

The procedure fetch(n) fetches n octets relative to the program counter, incrementing it by the same number. It is used both to fetch instructions and to fetch immediate operands.

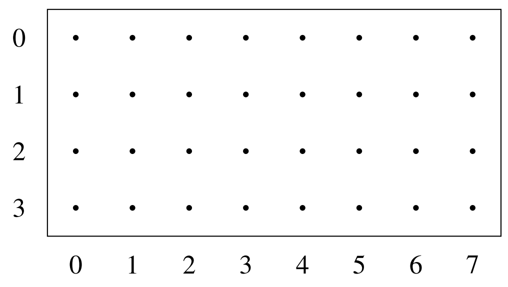
```
fetch($n$) ≡ var $x := \mathrm{load}(n,\ PC)$
             $PC := (PC + n) \bmod 2^{64}$
             return $x$                                            (5)
```

E. DEVICE ACCESS PROCEDURES

This section describes the procedures for device access. Since the devices interface the machine with the real world, the semantics can be described only informally.

1) IMAGE INPUT

The *Image Input* device allows the machine to consume an image as a two-dimensional array of points of light intensity values. The following figure shows an example of such an array, consisting of 32 sampling points arranged in 8 columns and 4 rows.



As shown, both columns and rows are numbered consecutively, starting at 0. The spacing between the sampling points must be uniform in both horizontal and vertical directions, and an anti-aliasing filter must be employed to limit the bandwidth of the image to satisfy the Nyquist-Shannon sampling theorem.

Each picture element detects the intensity of light transmitted or reflected at a sampling point in that particular position of the image, represented as one of 256 intensity levels, from 0 (minimum intensity) to 255 (maximum intensity). Values between 0 and 255 represent intermediate intensities between these extremes.

Definition 16 (Read frame): The readframe() operation reads a new frame and returns the number of columns, c, and the number of rows r.

$$(c, r) = \text{readframe}()$$

Definition 17 (Read pixel): The readpixel() operation returns the intensity, z, of the point at column x and row y.

$$z = \text{readpixel}(x, y)$$

2) IMAGE OUTPUT

The *Image Output* device allows the machine to produce an image represented as a two-dimensional array of points of color space values. Moving images can be produced as a sequence of images.

Definition 18 (New frame): The newframe() operation finishes and renders the frame constructed so far, and it sets the width of the next frame to w, the height to h, and the sample rate to r.

$$\text{newframe}(w, h, r)$$

Definition 19 (Set pixel): The setpixel() operation sets the red value to r, the green value to g, and the blue value to b for the pixel at column x and row y.

$$\text{setpixel}(x, y, r, g, b)$$

3) AUDIO OUTPUT

The *Audio Output* device allows the machine to produce a two-channel audio signal encoded digitally using Linear Pulse Code Modulation. The device must create an audio signal passing through a series of magnitude values specified by the program. The bandwidth of this audio signal must be less than half of the sampling frequency. Each channel value is in the range $\{0, \ldots, 2^{16} - 1\}$.

Definition 20 (Add sample): The addsample() operation sets the audio signal magnitude of the left channel to l and the one of the right channel to r.

$$\text{addsample}(l, r)$$

4) TEXT OUTPUT

The *Text Output* device allows the machine to produce a stream of text.

Definition 21 (Put character): The putchar() operation produces the character with Unicode code point c.

$$\text{putchar}(c)$$

5) OCTET OUTPUT

The *Text Output* device allows the machine to produce a stream of 8-bit numbers.

Definition 22 (Put byte): The putbyte() operation produces the octet x.

$$\text{putbyte}(x)$$

F. INSTRUCTION SEMANTICS

Table 3 summarizes the instruction semantics. The instruction cycle proceeds as follows:

1) Execute c := fetch(1), and locate the table entry whose "Hex" column value is c.
2) Execute x := fetch(n) for every variable, (n)x, in the "Immediate" column of the entry.
3) Execute v := pop() for every variable v listed in the "Pop" column of the entry.
4) Execute all operations listed in the "Explicit effects" column of the entry.
5) Execute push(e) for every expression e listed in the "Push" column of the entry.

This cycle is repeated until $T = 1$.

APPENDIX B
MORE ON THE ASSEMBLY LANGUAGE

As mentioned in Section III-D1, the language of the assembler is richer than the instruction set of the abstract machine, see Table 4. The assembler comes with a set of integration tests, each consisting of a small assembly program and the expected final stack. Adding more such tests would increase the trust in the assembler, but we do not yet have a complete specification of the assembly language to guide this work.

Somewhat related, we also wanted to prove formally how more complex "pseudo-instructions" can safely be implemented using the native instructions of our abstract machine. While conceptually clear, the mathematical description discussed in Section II-C is not well suited for such proofs. Thus, a second mathematical description of the abstract machine was produced from scratch, which assumes that the reader has a background in theoretical computer science and an

TABLE 3. Instruction semantics.

Hex	Mnemonic	Comment	Immediate	Pop	Explicit effects	Push
00	EXIT	Stop execution	–	–	$T := 1$	–
01	NOP	No operation	–	–	–	–
02	JUMP	Jump to address	–	a	$PC := a$	–
03	JZ_FWD	Jump forward on zero	$(1)d$	x	$PC := PC + \text{if}(x = 0, d, 0)$	–
04	JZ_BACK	Jump backward on zero	$(1)d$	x	$PC := PC - \text{if}(x = 0, d + 1, 0)$	–
05	SET_SP	Set stack pointer	–	a	$SP := a$	–
06	GET_PC	Get program counter	–	–	–	PC
07	GET_SP	Get stack pointer	–	–	–	SP
08	PUSH0	Push literal zero	–	–	–	0
09	PUSH1	Push 1 immediate octet	$(1)x$	–	–	x
0A	PUSH2	Push 2 immediate octets	$(2)x$	–	–	x
0B	PUSH4	Push 4 immediate octets	$(4)x$	–	–	x
0C	PUSH8	Push 8 immediate octets	$(8)x$	–	–	x
10	LOAD1	Load 1 memory octet	–	a	–	$\text{load}(1, a)$
11	LOAD2	Load 2 memory octets	–	a	–	$\text{load}(2, a)$
12	LOAD4	Load 4 memory octets	–	a	–	$\text{load}(4, a)$
13	LOAD8	Load 8 memory octets	–	a	–	$\text{load}(8, a)$
14	STORE1	Store 1 memory octet	–	a, x	$\text{store}(1, a, x)$	–
15	STORE2	Store 2 memory octets	–	a, x	$\text{store}(2, a, x)$	–
16	STORE4	Store 4 memory octets	–	a, x	$\text{store}(4, a, x)$	–
17	STORE8	Store 8 memory octets	–	a, x	$\text{store}(8, a, x)$	–
20	ADD	Add	–	y, x	–	$\text{add}(x, y)$
21	MULT	Multiply	–	y, x	–	$\text{mul}(x, y)$
22	DIV	Divide	–	y, x	–	$\text{div}(x, y)$
23	REM	Find remainder	–	y, x	–	$\text{rem}(x, y)$
24	LT	Less than	–	y, x	–	$\text{if}(x < y, -1, 0)$
28	AND	Bitwise "and"	–	y, x	–	$\text{and}(x, y)$
29	OR	Bitwise "or"	–	y, x	–	$\text{or}(x, y)$
2A	NOT	Bitwise "not"	–	x	–	$\text{not}(x, y)$
2B	XOR	Bitwise "exclusive or"	–	y, x	–	$\text{xor}(x, y)$
2C	POW2	Binary power	–	x	–	$\text{pow2}(x)$
F9	PUT_BYTE	Put byte	–	x	$\text{putbyte}(x)$	–
FA	PUT_CHAR	Put character	–	c	$\text{putchar}(c)$	–
FB	ADD_SAMPLE	Put audio sample	–	r, l	$\text{addsample}(l, r)$	–
FC	SET_PIXEL	Put pixel	–	b, g, r, y, x	$\text{setpixel}(x, y, r, g, b)$	–
FD	NEW_FRAME	Output frame	–	r, h, w	$\text{newframe}(w, h, r)$	–
FE	READ_PIXEL	Get pixel	–	x, y	$z := \text{readpixel}(x, y)$	z
FF	READ_FRAME	Input frame	–	–	$(c, r) := \text{readframe}()$	c, r

understanding of Coq, but the specifications of each instruction are still recognizable:

```
Equations oneStep' (op: Z) : M unit :=
  ...
  oneStep' JZ_FWD :=
    let* o := next 1 in
    let* x := pop64 in
    (if (decide (x = 0 :> Z))
     then
       let* pc := get' PC in
       put' PC (offset o pc)
     else
       ret tt);
  ...
```

We have only proven some basic properties of the machine so far. The fact that the machine has no separation between the program, stack and other memory has been a challenge. However, we have identified a number of useful concepts and strategies to deal with this, some inspired by separation logic [15], and the work is still ongoing.

APPENDIX C
COMPILER SUPPLEMENTARY ASPECTS

This annex complements some aspects of the compiler design discussed in the main text.

A. INSTRUCTION EXPANSION

Let us illustrate the expansion of an RTL rule with an example. Consider this canonical rule for the arithmetic division with 3 operands: operand0 ← operand1/operand2. Its expansion involves three steps: pushing the first operand, dividing by the second one and popping the result. This translates into three operations on TR: push (TR ← operand1), arithmetic operation (TR ← TR / operand2), and pop (operand0 ← TR). The corresponding RTL pattern would be like this one in the machine description:

```
(define_expand "udivdi3"
 [(set (match_operand:DI 0 "" "")
       (udiv:DI (match_operand:DI 1 "" "")
                (match_operand:DI 2 "" "")))]
  ""
{
  ivm_expand_push(operands[1], mode);
  rtx udiv_rtx = gen_rtx_UDIV(mode, TR_REG_RTX(mode),
                              operands[2]);
  emit_insn(gen_rtx_SET(TR_REG_RTX(mode), udiv_rtx));
  ivm_expand_pop(operands[0], mode);
  DONE;
})
```

Here functions ivm_expand_push pushes an operand and ivm_expand_pop pops the result. Once the result is popped, the content of the TR register is undefined. For this reason, the ivm_expand_pop function needs to clobber it.

This is the how the three-address based GIMPLE forms are translated into several one-operand instructions, with the help of the instrumental TR register. For this example, the RTL

TABLE 4. Assembly language grammar.

⟨assembly⟩	::=	⟨import⟩ ∗ ⟨statement⟩ ∗
⟨import⟩	::=	"IMPORT" (⟨id⟩ "/")+ ⟨id⟩
⟨statement⟩	::=	⟨id⟩ ":" | "EXPORT" ⟨id⟩ | ⟨id⟩ "=" ⟨expression⟩ | ⟨data⟩ | "space" ⟨expression⟩ | ⟨instruction⟩ "!"∗ ⟨expression⟩∗
⟨expression⟩	::=	⟨positive numeral⟩ | ⟨id⟩ | ("−" | "∼" | "$" | "&") ⟨expression⟩ | "(" ⟨operator⟩ ⟨expression⟩+ ")"
⟨operator⟩	::=	"+" | "*" | "&" | "|" | "^" | "=" | "<u" | "<s" | "<=u" | "<=s" | ">=u" | ">=s" | ">u" | ">s"
	|	"«" | "»u" | "»s" | "/u" | "/s" | "%u" | "%s"
⟨instruction⟩	::=	"exit" | "push" | "set_sp" | "jump" | "jump_zero" | "jump_not_zero" | "call" | "return"
	|	"load1" | "load2" | "load4" | "load8" | "sigx1" | "sigx2" | "sigx4" | "sigx8"
	|	"store1" | "store2" | "store4" | "store8" | "add" | "sub" | "mult" | "neg" | "and" | "or" | "xor" | "not"
	|	"pow2" | "shift_l" | "shift_ru" | "shift_rs" | "div_u" | "div_s" | "rem_u" | "rem_s"
	|	"lt_u" | "lt_s" | "lte_u" | "lte_s" | "eq" | "gte_u" | "gte_s" | "gt_u" | "gt_s"
	|	"read_frame" | "read_pixel" | "put_char" | "put_byte" | "new_frame" | "set_pixel" | "add_sample"
⟨data⟩	::=	("data1" | "data2" | "data4" | "data8") "[" ⟨expression⟩ ∗ "]"

expressions emitted by the expansion for a transfer such as `AR` ← `X1/X2`, with three general purpose registers, would be:

```
(set  (reg:DI  TR_REGNUM)
      (unspec_volatile:DI
           [(reg:DI  TR_REGNUM)
            (reg:DI  X1_REGNUM)]  UNSPEC_PUSH_TR))
(set  (reg:DI  TR_REGNUM)
      (udiv:DI  (reg:DI  TR_REGNUM)
                (reg:DI  X2_REGNUM)))
(set  (reg:DI  AR_REGNUM)
      (unspec_volatile:DI
            [(reg:DI  TR_REGNUM)]  UNSPEC_POP_TR))
(clobber  (reg: DI  TR_REGNUM))
```

Finally, after all the compilation passes, the resulting RTL expressions must be printed as assembly instructions. Rules to output the assembly also need to be defined in the machine description file. For example, next RTL pattern describes how to write the assembly for action `TR` ← `TR / operand2`:

```
(define_insn "udivdi1"
[(set  (match_operand:DI  0
         "tr_register_operand"  "=r,r")
      (udiv:DI  (match_operand:DI  1
                  "tr_register_operand"  "r,r")
               (match_operand:DI  2
                   "arithmetic_operand"  "i,rm"
                   )))]
    ""
    "@
    div_u!  %2
    load8!  %2\;div_u"
)
```

Here it is defined which assembly instructions will be written when finding an RTL expression that matches this rule. The string `tr_register_operand` is a *predicate*, a function that is only true for the TR register. Quoted letters represents *constraints* associated to addressing modes (`"i"` for immediate, `"r"` for register and `"m"` for memory). Predicates and target-specific constraints are also defined as part of the machine description.

B. ABI

The ABI (Application Binary Interface) is a key piece for the integration of program modules written in C (and compiled with the C compiler) and hand-written assembly code. A summary of the designed ABI is included in this section.

a: DATA LAYOUT

Integer data can be of four sizes: 1, 2, 4, or 8 bytes long. For signed integers, the corresponding types are `char`, `short`, `int` and `long`. There are the corresponding unsigned types of the same size. For real numbers, the type `float` is 4 bytes long while the type `double` is 8 bytes long. Two additional 128-bit data types are supported by the compiler, that are mapped into 2 consecutive locations: `long128` and `complex double`. In the case of structures, it should be noted that a padding may be added between structure members, or at the end of the aggregate type, but never at the beginning, before the first member.

b: CALLING CONVENTION

A unique ABI is used for all functions regardless the number of arguments (no arguments, a fixed number of arguments or a variable number of arguments). The calling convention is based on the *caller-pops-arguments* rule, so that the caller is in charge of moving the return value to its destination and releasing the arguments. In short, this is the sequence of actions followed when invoking a function:

1) The caller may allocate one or two stack slots for storing the return value. Currently, the compiler uses the register AR to place 64-bit return values (or smaller), and when returning a 128-bit value, the pair (X1,AR) is used, being AR the less significant 64-bit word.
2) The caller passes all arguments on the stack
3) The caller invokes the function (call instruction).
4) After the function returns, the return value has been placed by the callee in the stack position immediately above the return address. At this point, the caller copies the return value (for example, to AR) and releases arguments, if necessary.

An example of the assembly code generated for a C code calling a function is shown in Fig. 6.

```
int
foo  (unsigned char a,
      unsigned int b,
      unsigned long c)
{
    return a & b & c;
}

main()
{
    return foo(1, 10, 100);
}
```

```
        EXPORT  foo
foo:
        set_sp! &-16  # prologue: allocate stack mapped GPRs

        load1! &17  # get arguments
        load4! &19
        load4! &21
        and
        and
        store4! &18 # write return value on the last argument

        set_sp! &16 # epilogue: free stack mapped regs.
        return

        EXPORT  main
main:
        set_sp! &-16 # prologue

        push! 100  # push arguments
        push! 10
        push! 1

        call! foo  # call

        store8! &3   # caller pops the return value into AR
        set_sp &2    # caller frees the remainder arguments

        load4! &0    # get AR
        store4! &18  # write return value on the last argument

        set_sp! &16  #epilogue
        return
```

FIGURE 6. **Example of a function call (C code on the left; generated IVM assembly on the right).**

C. LIBRARIES

The GCC compiler needs to be accompanied by the standard C libraries in order to produce fully functional code. Two key libraries have been ported to the architecture of the abstract machine:

- The *libgcc* library, which includes a collection of routines for floating point emulation (`libgcc.a`). It is necessary as the abstract machine supports only native integer arithmetic (floating-point arithmetic is software emulated).
- The *newlib* library [16], [17], which provides the C standard library: `crt0.o` (the C runtime startup file), `libc.a` (the standard C library), and `libm.a` (the mathematical C library).

The C run-time startup file is linked by default as the first object of the executable program, unless a specific entry point is defined (`-e` command-line flag). Notice that the design of `crt0` is very dependent on how the implementation of the abstract machine builds and initializes the binary code. The `crt0` provided with newlib is responsible for: (1) providing a default start point through the global label `_start`, (2) initializing a global variable pointing to the beginning of the heap, (3) initializing main arguments `argc` and `argv`, (4) calling the function `main(int argc, char* argv[])`, and (5) on exit, leaving the `main()` return value on the stack.

Two classes of libraries, static (`.a`) and dynamic (`.so`) are supported by the compiler toolchain. The terms *static* and *dynamic* refer to the format of the library rather than how they are loaded. The whole compilation process is static, that is, all used functions are included in the final binary, as no dynamic loader exists. The `ld` script separates different objects included in a library in order to process them together with the other objects coming from C sources. The `ld` script is able to reduce the size of the output by selecting only those objects in libraries that are actually used by the C sources being linked.

D. OPTIMIZATIONS

The assembly code generated by the compiler benefits from three kinds of optimizations: (1) the optimization passes provided by GCC, (2) specific peepholes for the abstract machine target, and (3) some lower-level optimizations at assembly level. By default, the compiler has been configured to use the `-O2` optimization level, which offers a good balance between speed (executed instructions) and size. The peephole transformations are a key element in optimization. Peepholes are applied as another GCC optimization pass, but they are target dependent. For the abstract machine, a wide set of peephole transformations has been defined. Finally, a set of assembly level optimizations has been considered in the output phase.

E. VALIDATION

In addition to the specific applications related with this project, such as the boxing library and the supported format renderers (PDF, JPEG, TIFF,...), the compiler has been tested with several C compiler testsuites including:

- the GCC testsuite for C, distributed along with GCC. It can be launched after building the compiler with `make check-gcc-c`. This is a very large collection of basic tests (+85000 tests performed), including the so-called torture tests, specially designed to stress the compiler,

- the TCC test suite [18], which contains nearly one hundred programs testing basic features of the C language,
- a collection of benchmarks from the LLVM testsuite [19]: *Olden, Prtdist, VersaBench, Fhourstones, lemon, llubenchmark, mafft, nbench, oggenc, spiff, viterbi, SMG2000, McGill, Shootout, Stanford*, and *CoyoteBench*.

One of the challenges when testing real programs has been the lack of file systems for the abstract machine. An approach to tackle this limitation, is including the files in memory data structures in the C source, in such a way the access to files is emulated by accessing such structures. This approach involves modifying the source code. For cases like the code from the LLVM testsuite, an in-memory folderless file system generator [20] was developed. It generates a C file with the contents of the filesystem together the basic low-level file primitives (`open, close, read, write, ...`). In this way, this generated file can be linked with the rest of source files, which can access the files without requiring modifications. These features rely on the linker's ability to supersede the libc file primitives by those provided together with the filesystem.

ACKNOWLEDGMENT

Joschua Thomas Simon-Liedtke participated in the experiment (see Section VI-A) and wrote a partial machine implementation. This resulted in feedback on the guide in Section II-A and led to its improvement.

REFERENCES

[1] The Coq Development Team. *The Coq Proof Assistant 8.12*. Zenodo. Accessed: Jul. 2020. [Online]. Available: https://coq.inria.fr/

[2] M. Sozeau and C. Mangin, "Equations reloaded: High-level dependently-typed functional programming and proving in Coq," *Proc. ACM Program. Lang.*, vol. 3, pp. 1–29, Jul. 2019.

[3] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with web assembly," in *Proc. 38th Conf. Program. Lang. Design Implement. (ACM SIGPLAN)*, A. Cohen and M. T. Vechev, Eds., Barcelona, Spain, Jun. 2017, pp. 185–200.

[4] Stephan Tolksdorf. *FParsec (1.1.0)*. Accessed: Jan. 5, 2020. [Online]. Available: https://www.quanttec.com/fparsec/

[5] H. Gunnarsson and T. Lundqvist, "Porting the GNU C compiler to the Thor microprocessor," M.S. thesis, Göteborg, Sweden, 1995. [Online]. Available: http://tlundqvist.org/Publications/

[6] R. Lorie, "Long term preservation of digital information," in *Proc. JCDL*, 2001, pp. 346–352.

[7] L. T. Nguyen and A. Kay, "The cuneiform tablets of 2015," in *Proc. ACM Int. Symp. New Ideas, New Paradigms, Reflections Program. Softw. (Onward)*, G. C. Murphy Ed., Pittsburgh, PA, USA, Oct. 2015, pp. 297–307.

[8] V. Joguin, "Passive digital preservation now & later: Microfilm, micr'olonys and dna," in *Proc. 16th Int. Conf. Digit. Preservation*, B. Sierman and A. Puggioni, Eds., 2019, pp. 1–6.

[9] R. Appuswamy and V. Joguin, "Universal layout emulation for long-term database archival," in *Proc. 11st Conf. Innov. Data Syst. Res. (CIDR)*, Jan. 2021, pp. 1–6.

[10] J. Braun, J. Buchmann, D. Demirel, M. Fujiwara, M. Geihs, S. Moriai, M. Sasaki, and A. Waseda, "LINCOS—A storage system providing long-term integrity, authenticity, and confidentiality (full paper)," IACR Cryptol. ePrint Arch., Tech. Rep. 2016/742, 2016, p. 742. [Online]. Available: https://eprint.iacr.org/2016/742

[11] R. M. Stallman, *The GCC Developer Community*, document GNU Compiler Collection Internals (for GCC Version 10.2.0), 2020. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-10.2.0/gccint.pdf

[12] K. Walfridsson. (Dec. 1, 2020). *Writing a GCC Back End*. Accessed: 2017. [Online]. Available: https://kristerw.blogspot.com/2017/08/writing-gcc-backend_4.html

[13] (Dec. 1, 2020). *Writing GCC Machine Descriptions*. Accessed: 2010. [Online]. Available: http://www.cse.iitb.ac.in/grc/intdocs/gcc-writing-md.html

[14] (Dec. 1, 2020). *ZPU—The Worlds Smallest 32 Bit CPU With GCC Toolchain*. Accessed: 2009. [Online]. Available: https://opencores.org/projects/zpu

[15] P. O'Hearn, "Separation logic," *Commun. ACM*, vol. 62, pp. 86–95, Jan. 2019.

[16] C. Vinschen and J. Johnston. (Mar. 1, 2021). *The Red Hat NewLib C Library*. Accessed: 2021. [Online]. Available: https://sourceware.org/newlib/

[17] J. Bennett. (Mar. 1, 2021). *Howto Porting NewLib: A Simple Guide*. Accessed: 2010. [Online]. Available: https://www.embecosm.com/appnotes/ean9/ean9-howto-newlib-1.0.html

[18] F. Bellard. (Mar. 1, 2021). *Tiny C Compiler*. [Online]. Available: http://bellard.org/tcc

[19] *LLVM Testsuite*. Accessed: Nov. 2021. [Online]. Available: https://github.com/llvm/llvm-test-suite

[20] E. Gutierrez, S. Romero, and O. Plata. (2020). *IVM Filesystem Generator*. [Online]. Available: https://github.com/immortalvm/ivm-fs

IVAR RUMMELHOFF received the dr.scient. degree in mathematical logic. He has a background in the software industry. He is currently a Senior Research Scientist with the Norwegian Computing Center working in the area of digital transformation.

ELADIO GUTIÉRREZ received the M.Sc. and Ph.D. degrees in telecommunication engineering from the University of Malaga, Spain, in 1995 and 2001, respectively. Since 2003, he has been an Associate Professor with the Department of Computer Architecture, University of Malaga.

THOR KRISTOFFERSEN received the dr.scient. degree in computer science from the University of Oslo, in 1998. Since then, he has been working with the Norwegian Computing Center. He is currently a Senior Research Scientist with the Norwegian Computing Center.

OLE LIABØ received the bachelor's degree in computer science from the Trondheim College of Engineering, in 1996. He is currently the Innovation Manager with Piql AS.

BJARTE M. ØSTVOLD received the dr.ing. degree from the Norwegian University of Science and Technology, Trondheim, Norway, in 1999. Since 1999, he has been working with the Norwegian Computing Center, Oslo, Norway.

OSCAR PLATA received the Ph.D. degree in physics from the University of Santiago de Compostela, Spain, in 1989. He worked as an Assistant Professor with the University of Santiago de Compostela, where he became an Associate Professor, in 1990. He moved to the University of Malaga, in 1995, where he became a Full Professor with the Department of Computer Architecture, in 2002. His research interests include high performance computing and parallel architectures. He was an Associate Editor of IEEE TRANSACTIONS OF COMPUTERS, from 2015 to 2019.

SERGIO ROMERO received the M.Sc. and Ph.D. degrees in computer science from the University of Malaga, Spain, in 1996 and 2000, respectively. Since 2003, he has been an Associate Professor with the Department of Computer Architecture, University of Malaga.

• • •