Immortal Virtual Machine project
The IVM64 C Compiler
Technical Report

Eladio Gutiérrez Carrasco Óscar Plata González Sergio Romero Montiel

Department of Computer Architecture, University of Málaga

June 15, 2021

Contents

1 Introduction **2**

2 GNU Compiler Collection (GCC) **2**
2.1 The `cc1` program . . . 2
2.1.1 Front end . . . 3
2.1.2 Middle end . . . 3
2.1.3 Back end . . . 4

3 Related work **4**

4 The target IVM64 **5**
4.1 IVM64 Instruction Set Architecture . . . 5
4.2 Assembly syntax . . . 6
4.2.1 Restrictions to the assembler . . . 9
4.3 Stackification scheme . . . 9
4.4 Instruction expansion . . . 12
4.5 Application Binary Interface (ABI) . . . 13
4.5.1 Data layout . . . 13
4.5.2 Calling convention . . . 15
4.6 Inline assembly . . . 18
4.7 Compiler tool-chain . . . 21
4.8 External tools . . . 23
4.9 Libraries . . . 25
4.9.1 Startup file `crt0` . . . 26
4.9.2 Function `alloca` . . . 26
4.9.3 Functions `setjmp/longjmp` . . . 27
4.9.4 Optimized string functions . . . 27
4.10 Reserved symbols . . . 28
4.11 Optimizations . . . 28
4.12 Using the compiler . . . 29
4.13 Structure of the target . . . 29

5 Validation **30**

6 Conclusions **32**

1 Introduction

The Immortal Virtual Machine (iVM) project [1] is aimed at creating an information preservation solution that ensures the access to today's digital data in the future (over hundreds of years). With this goal, an abstract virtual machine is proposed to reconstruct the preserved data. Data is stored in a preservation medium using the film technology developed by Piql [2], where information is digitally coded in frames in a reel of film.

The role of the iVM is to implement the format decoders that are stored in binary format along with the data to be preserved. This results in self-executing contents which are able to produce consistent output (images, sounds, ...) on the output devices. Two important requirements of this abstract machine are to be easily described and easy to implement by future developers.

One of the key points of this approach is the availability of a compiler that is in charge of generating the iVM code from the decoder source codes. Theses decoders are assumed to be written in a very consolidated language like C. Since many C compiler infrastructures are available, we need to choose the most suitable, and selecting an open-source one has been a major decisive factor. Among the available options, several reasons led us to adopt GCC (GNU Compiler Collection) [3]: it is a widespread full C compiler, it has a great community support and a very stable design API between different versions.

This document describes the most relevant aspects of the compiler design and its usage. The compiler's mission is to provide an assembly representation of the C programs, that eventually will be converted to binary by an assembler. The assembly language serves as an interface between the compiler and the assembler. Actually, the developed compiler is a C cross-compiler, to be compiled and executed on today's computers.

Hereinafter the term IVM64 refers to the architecture of the abstract virtual machine for which the compiler is targetted.

2 GNU Compiler Collection (GCC)

GCC is a compiler system supporting several programming languages including C, C++, Fortran, Ada, etc, that have been ported to an extensive set of targets (x86, ARM, MIPS, ...), most of them register-based architectures. The term *target* refers to the processor for which the compiler generates the output code.

Building an executable program from C source files is performed in a few consecutive steps. In fact, the program `gcc` behaves as a driver invoking the necessary tools to carry out these steps. First, each source is preprocessed by the `cpp` program. Then, the preprocessed sources are translated to assembly by the program `cc1` (the actual C compiler). The generated assembly codes are translated to object files (a binary linkable format) by the assembler (the `as` program). Finally all the objects are combined together with the required libraries (which are really collections of objects) into the final binary executable by the linker (the `ld` program). This sequence of tools is commonly known as the compiler *toolchain*.

When GCC is built, only the `cpp` and `cc1` programs are produced. The other tools are external, and must be provided in order to complete the full toolchain. Likewise, a full GCC distribution should include a minimal set of libraries so as to produce fully functional executables: *libc* (the standard C library), *libm* (the mathematical C library) and *libgcc* (which provides, for example, software floating point arithmetic in platforms lacking it natively).

The rest of this section is focused on the `cc1` program which is the core of the compilation system.

2.1 The `cc1` program

The C compiler `cc1` is the program in charge of translating the C language into the target assembly language. This process is organized in three stages: the *front end* (which translates from C into an intermediate representation), the *middle end* (that performs the main optimization process with dozens to hundred of passes) and the *back end* (which outputs the optimized representation of the program

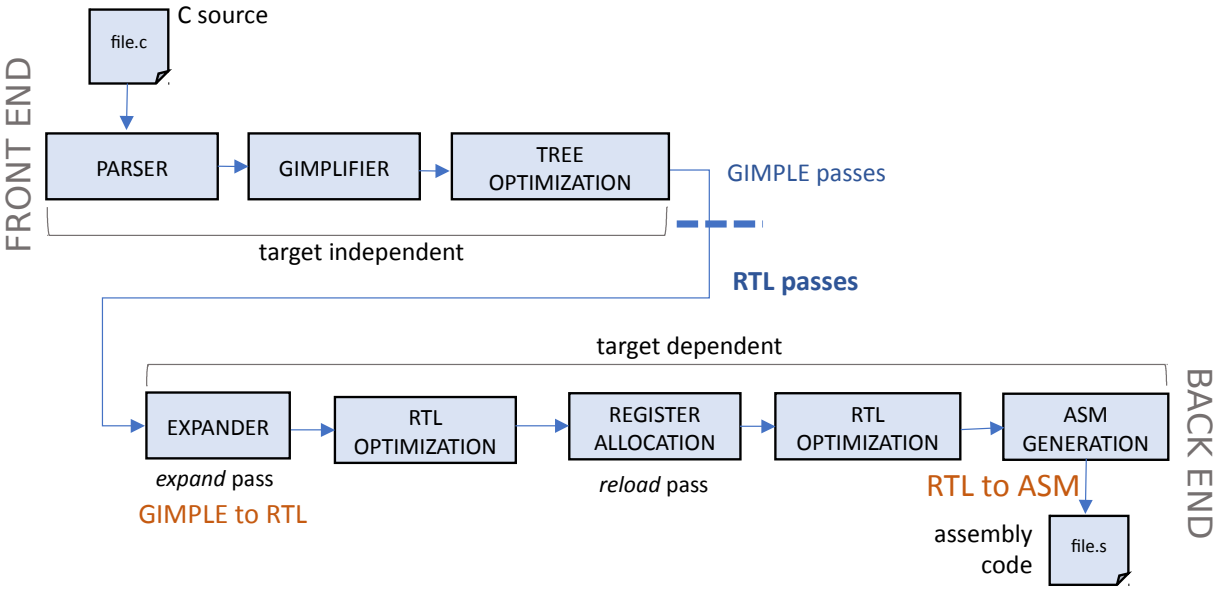

Figure 1: Overview of `cc1` stages (some boxes like optimizations may involve many compiler passes).

into assembly language). Due to its design, all languages supported by GCC use the same intermediate language and so all GCC compilers share the same optimization algorithms. An overview of different compilation stages is shown in figure 1.

2.1.1 Front end

The C front end design is a recursive-descent parser. This stage parses the source file producing an abstract syntax tree which is independent of the target. GCC uses two forms of abstract syntax trees: GENERIC and GIMPLE. Several optimization passes use these abstract tree representations. The former is a more complex representation, a construct in GENERIC may produce multiple GIMPLE instructions. The C front ends produce GENERIC directly in the front end, then the so-called *gimplifier* converts GENERIC into the simpler SSA-based GIMPLE form (three-address code using temporary variables).

2.1.2 Middle end

The GCC middle end performs the big part of the code analysis and optimization, working on representations that are independent of both the compiled language and the target architecture.

It starts after the syntax tree of a C function is completed. After optimizing the GIMPLE tree, the GIMPLE form is translated into an intermediate internal representation, the so-called Register Transfer Language (RTL). This translation process is known as *expansion*. The expansion gives rise to a sequence of RTL expressions (RTX) for the function, that can be seen as a coarse approach of what will the final assembly code.

An RTX is a sort of Lisp expression. The RTL representation after each RTL pass can be dumped in a textual form using some command line options (see section 4.12). Figure 2 shows the RTX sequence for a function. This sequence is target dependent and the one shown is for the ft32 target. The RTL representation is a numbered list of expressions, where each one has an index pointing to its next and previous one.

Observe that 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. This involves to reference additional registers with no correspondence to real registers. These are the so-called *pseudoregisters*. Pseudoregisters need to be assigned to real registers (if any in the target processor) or to stack positions. This operation is known as *register allocation*, which is carried out by the *reload* pass (see figure 1).

```
int a,b,c;
void main(){
  c = a * b;
}
```

(a)

```
void main ()
{
  int a.0_1;
  int b.1_2;
  int _3;

  <bb 2>:
  a.0_1 ={v} a;
  b.1_2 ={v} b;
  _3 = a.0_1 * b.1_2;
  c ={v} _3;
  return;
}
```

(b)

```
(note 1 0 3 NOTE_INSN_DELETED)
(note 3 1 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(note 2 3 5 NOTE_INSN_FUNCTION_BEG)
(insn 5 2 6 (set (reg:SI 42) ;; [a.0_1]
                 (mem:SI (symbol_ref:SI ("a"))))
(insn 6 5 7 (set (reg:SI 43) ;; [b.1_2]
                 (mem:SI (symbol_ref:SI ("b"))))
(insn 7 6 8 (set (reg:SI 44) ;; [_3]
                 (mult:SI (reg:SI 42)
                          (reg:SI 43 ))))
(insn 8 7 11 (set (mem:SI (symbol_ref:SI ("c"))
                  (reg:SI 44))))
```

(c)

Figure 2: Textual forms of the GCC internal representations for a register-based target: (a) C source, (b) GIMPLE and (c) RTL.

Apart from the crucial reload pass, most of the passes in this stage perform optimizations for improving the performance of the generated assembly. The exact set of GCC optimizations depends on the GCC release, and it includes many standard algorithms [3].

2.1.3 Back end

Strictly speaking, this stage corresponds with the *final* pass, that converts the RTX sequence into assembly statements. Nevertheless, all the RTL passes are target-dependent, and the RTL expressions generated in any of these passes must meet the *machine description* of the target architecture (see section 4.13).

This description specifies the features of the architecture like the word size, endianness, the direction towards which the stack grows, calling convention, etc. A machine description defines valid RTL patterns, fulfilling certain constrains such as the addressing modes, as well as which assembly instructions are going to be generated for each pattern. As the compilation nears to its end, only valid RTL patterns and real machine registers should appear in the RTL representation, in such a way the final pass can print all the RTXs as target assembly constructions.

As the target object of this project is a pure stack machine with no registers, in this stage, all register references are transformed into memory references in the stack. This *stackification* scheme is described in section 4.3.

3 Related work

A good starting point when porting GCC to a new processor is studying the existing targets, either ones within the GCC distribution or others from third-party sources. Finding a target with similar features can be very helpful in the design. The set of architectures, brand new and old ones, to which GCC has been ported is extensive. However, most of them are processors with some kind of general purpose registers. In fact no one in the standard GCC distribution has no registers.

There is not any official guide of how porting GCC, apart from the GCC reference [3] that describes the internal aspects of the compiler. Some guidelines can be found in references like [4, 5], where the process of creating a new target for a register-based architecture is explained in a general way. Also some other ports for machines using registers, like [6, 7], contribute some details about how to porting GCC. Focusing on pure stack machines, the number of targets is scarce. We can highlight two of them: Thor [8] and ZPU [9] processors.

The Thor processor is a 32-bit CPU developed for aerospatial applications. It is a relatively old project, where a GCC back end for this processor was developed (version GCC 2.7 was used, released in 1995). The document describing the design of the compiler [8] is an excellent guide about porting a new

target. The processor is a pure stack machine with no registers other than the PC and SP, although the architecture is slightly more complex than IVM64 (more instructions, execution pipeline, ...). The Thor project contributes some valuable ideas of how handle the lack of registers.

ZPU is another small zero-operand 32-bit processor with a simplified architecture. There is no document describing design of the GCC back end, but the sources codes are available in [9] (for GCC version 3.4.2, released in 2004). These sources include not only the compiler but the full tool-chain, together with the assembler, the linker and other helper programs. Unlike Thor that makes the stackification of operands as soon as possible (in the expansion pass), the ZPU backend postpones it to the final pass.

4 The target IVM64

The Immortal Virtual Machine is a 64-bit abstract processor (IVM64) designed with a goal, to be easily described in a formal way, so that future developers will be able to recreate it and execute the software necessary to extract the information preserved in the reel together with the software itself [10]. IVM64 is a stack machine that includes only these elements:

- a byte-addressable memory of arbitrary size
- two specific-purpose registers: the Program Counter (PC) and the Stack Pointer (SP)
- a halt bit

Immediately after the machine powers on, PC points to the first memory position (address 0), SP points to the last one and the halt bit is reset. In this machine: the stack grows downwards, registers are 64-bit wide, the memory is byte-addressable and data are stored following a little-endian scheme. The execution stops with the `exit` instruction which enables the halt bit.

From the viewpoint of the development of a C compiler, it is necessary to know what the compiler must produce. As a design decision, the C compiler for IVM64 will be in charge of generating an assembly code for the target, in such a way this assembly code must be translated later to IVM64 binaries in terms of the IVM's native instructions.

The C compiler described in this document is actually a crosscompiler, that must be compiled and executed in today's computers (for example an x86-64 based PC) to produce an IVM64 assembly code. An external assembler tool to assemble the compiler output is required, i.e., to translate the output to the IVM64 native binary. In the context of this projects this tool was named *ivm implementation*. In addition to it, today's developer will need also another tool to simulate the binary after being assembled for testing and debugging purposes.

The instruction set and assembly syntax are briefly described next. In the one hand we find the native IVM64 instruction set that has a direct correspondence with the binary code; and in the other hand, we have the assembly syntax used in assembly programs. The crosscompiler never generates binary code, but assembly. A more complete description of both the assembly syntax and IVM64 instruction set can be found in [11, 12].

We refer to the C crosscompiler for the IVM64 target as `ivm64-gcc`.

4.1 IVM64 Instruction Set Architecture

Table 1, extracted from [12], summarizes the IVM64 instruction set architecture (ISA). Native instructions are classified in `exit` (stops the execution), `nop` (do nothing), flow control reading or writing the PC register (`jump`, `jz_fwd`, `jz_back` and `get_pc`), stack management reading or writing the SP register (`get_sp` and `set_sp`), data movement (`push<N>`, `load<N>` and `store<N>` (opcode in range [8..23]), unsigned arithmetic (opcode in range [32..44]) and data output (opcode in range[0xf9..0xff]).

The size of all instruction opcodes is 1 byte. Only conditional jumps (`jz_fwd`, `jz_back`) and `push<N>` instructions have one immediate operand. This operand is 1-byte sized for conditional jumps and `N` bytes in the case of `push<N>` instructions.

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	–	load(1, a)
11	LOAD2	Load 2 memory octets	–	a	–	load(2, a)
12	LOAD4	Load 4 memory octets	–	a	–	load(4, a)
13	LOAD8	Load 8 memory octets	–	a	–	load(8, a)
14	STORE1	Store 1 memory octet	–	a,x	store(1, a, x)	–
15	STORE2	Store 2 memory octets	–	a,x	store(2, a, x)	–
16	STORE4	Store 4 memory octets	–	a,x	store(4, a, x)	–
17	STORE8	Store 8 memory octets	–	a,x	store(8, a, x)	–
20	ADD	Add	–	y,x	–	add(x,y)
21	MULT	Multiply	–	y,x	–	mul(x,y)
22	DIV	Divide	–	y,x	–	div(x,y)
23	REM	Find remainder	–	y,x	–	rem(x,y)
24	LT	Less than	–	y,x	–	if$(x < y, -1, 0)$
28	AND	Bitwise “and”	–	y,x	–	and(x,y)
29	OR	Bitwise “or”	–	y,x	–	or(x,y)
2A	NOT	Bitwise “not”	–	x	–	not(x,y)
2B	XOR	Bitwise “exclusive or”	–	y,x	–	xor(x,y)
2C	POW2	Binary power	–	x	–	pow2(x)
F9	PUT_BYTE	Put byte	–	x	putbyte(x)	–
FA	PUT_CHAR	Put character	–	c	putchar(c)	–
FB	ADD_SAMPLE	Put audio sample	–	r,l	addsample(l,r)	–
FC	SET_PIXEL	Put pixel	–	b,g,r,y,x	setpixel(x,y,r,g,b)	–
FD	NEW_FRAME	Output frame	–	r,h,w	newframe(w,h,r)	–
FE	READ_PIXEL	Get pixel	–	x,y	$z :=$ readpixel(x,y)	z
FF	READ_FRAME	Input frame	–	–	$(c,r) :=$ readframe()	c,r

Table 1: IVM64 instruction set (from [12]).

Most instructions have their operands on the top of the stack (one or two words) and leave the result on it. So in the following, the expression *pop an operand* means reading the word on top of the stack (TOS) and increasing in 8 (size in bytes of a word) the the `SP` register; while the expression *push anything* means decrementing `SP` in 8 and writing in the new TOS a given value.

4.2 Assembly syntax

Assembly codes, such as those generated by `ivm64-gcc`, are not directly expressed in terms of native instruction mnemonics. Instead a richer syntax introduced by [13] is used, which includes mnemonics[1], directives, labels, abbreviations, operators and expressions. A semi-formal EBNF description of this syntax from [13] is as follows:

```
program = import* statement*;

import = "IMPORT" (identifier "/")+ identifier

statement = identifier ":"                                          (* label *)
          | "EXPORT" identifier                                     (* export declaration *)
          | identifier "=" expression                               (* abbreviation *)
          | "data1" "[" expression* "]" ("*" positive_numeral)?     (* data segment,  8 bits per value *)
          | "data2" "[" expression* "]" ("*" positive_numeral)?     (* data segment, 16 bits per value *)
          | "data4" "[" expression* "]" ("*" positive_numeral)?     (* data segment, 32 bits per value *)
```

[1] Some of the mnemonics have a direct correspondence with native instructions (e.g. `add`) but others are actually pseudoinstructions (e.g. `div_u`). Also some native instructions has no specific mnemonic, for example `get_sp` needs to be expressed as `push! &0`. Other mnemonics are mere aliases, like `return` that is equivalent to `jump`.

```
          | "data8" "[" expression* "]" ("*" positive_numeral)?    (* data segment, 64 bits per value *)
          | "space" expression                                    (* pointer static byte array *)

          | "exit" | "exit!" expression
          | "push" | "push!" expression | "push!!" expression expression | ...
          | "set_sp" | "set_sp!" expression
          | "jump" | "jump!" expression
          | "jump_zero" | "jump_zero!" expression | "jump_zero!!" expression expression
          | "jump_not_zero" | "jump_not_zero!" expression | "jump_not_zero!!" expression expression
          | "call" | "call!" expression
          | "return"                                              (* alias for "jump" *)

          | "load1" | "load1!" expression
          | "load2" | "load2!" expression
          | "load4" | "load4!" expression
          | "load8" | "load8!" expression
          | "sigx1" | "sigx1!" expression
          | "sigx2" | "sigx2!" expression
          | "sigx4" | "sigx4!" expression
          | "sigx8" | "sigx8!" expression                         (* no-op *)
          | "store1" | "store1!" expression | "store1!!" expression expression
          | "store2" | "store2!" expression | "store2!!" expression expression
          | "store4" | "store4!" expression | "store4!!" expression expression
          | "store8" | "store8!" expression | "store8!!" expression expression

          | "add" | "add!" expression | "add!!" expression expression
          | "sub" | "sub!" expression | "sub!!" expression expression
          | "mult" | "mult!" expression | "mult!!" expression expression
          | "neg" | "neg!" expression
          | "and" | "and!" expression | "and!!" expression expression
          | "or" | "or!" expression | "or!!" expression expression
          | "xor" | "xor!" expression | "xor!!" expression expression
          | "not" | "not!" expression
          | "pow2" | "pow2!" expression
          | "shift_l" | "shift_l!" expression | "shift_l!!" expression expression
          | "shift_ru" | "shift_ru!" expression | "shift_ru!!" expression expression
          | "shift_rs" | "shift_rs!" expression | "shift_rs!!" expression expression

          | "div_u" | "div_u!" expression | "div_u!!" expression expression
          | "div_s" | "div_s!" expression | "div_s!!" expression expression
          | "rem_u" | "rem_u!" expression | "rem_u!!" expression expression
          | "rem_s" | "rem_s!" expression | "rem_s!!" expression expression

          | "lt_u" | "lt_u!" expression | "lt_u!!" expression expression
          | "lt_s" | "lt_s!" expression | "lt_s!!" expression expression
          | "lte_u" | "lte_u!" expression | "lte_u!!" expression expression
          | "lte_s" | "lte_s!" expression | "lte_s!!" expression expression
          | "eq" | "eq!" expression | "eq!!" expression expression
          | "gte_u" | "gte_u!" expression | "gte_u!!" expression expression
          | "gte_s" | "gte_s!" expression | "gte_s!!" expression expression
          | "gt_u" | "gt_u!" expression | "gt_u!!" expression expression
          | "gt_s" | "gt_s!" expression | "gt_s!!" expression expression

          | "read_frame" | "read_frame!" expression
          | ... | "read_pixel!!" expression expression

          | ... | "put_char!" expression
          | ... | "put_byte!" expression
          | ... | "new_frame!!!" expression expression expression
          | ... | "set_pixel!!!!!" expression ...
          | ... | "add_sample!!" expression expression;

expression = positive_numeral  (* 0 to 2^64-1 *)
           | identifier        (* label or abbreviation *)

           | "-" expression    (* corresponding statement: neg *)
           | "~" expression    (* not *)
           | "$" expression    (* stack content *)
           | "&" expression    (* stack pointer *)

           | "(" "+" expression* ")"               (* add *)
           | "(" "*" expression* ")"               (* mult *)
           | "(" "&" expression* ")"               (* and *)
           | "(" "|" expression* ")"               (* or  *)
           | "(" "^" expression* ")"               (* xor *)

           | "(" "="   expression expression ")"    (* eq *)
           | "(" "<u"  expression expression ")"    (* lt_u *)
           | "(" "<s"  expression expression ")"    (* lt_s *)
           | "(" "<=u" expression expression ")"    (* lte_u *)
           | "(" "<=s" expression expression ")"    (* lte_s *)
           | "(" ">u"  expression expression ")"    (* gt_u *)
           | "(" ">s"  expression expression ")"    (* gt_s *)
           | "(" ">=u" expression expression ")"    (* gte_u *)
```

```
            | "(" ">=s" expression expression ")"    (* gte_s *)

            | "(" "<<"  expression expression ")"    (* shift_l *)
            | "(" ">>u" expression expression ")"    (* shift_r unsigned, unsigned *)
            | "(" ">>s" expression expression ")"    (* shift_r signed, unsigned *)
            | "(" "/u"  expression expression ")"    (* div_u *)
            | "(" "/s"  expression expression ")"    (* div_s *)
            | "(" "%u"  expression expression ")"    (* rem_u *)
            | "(" "%s"  expression expression ")"    (* rem_s *)

            | "(" "load1" expression ")"
            | "(" "load2" expression ")"
            | "(" "load4" expression ")"
            | "(" "load8" expression ")"
            | "(" "sigx1" expression ")"
            | "(" "sigx2" expression ")"
            | "(" "sigx4" expression ")"
            | "(" "sigx8" expression ")"             (* identity function *)

identifier = (letter | "_" | "." ) (letter | "_" | "." | digit)*;

An alternative notation for "immediate arguments"

    push* [ <e1> <e2> ... <en> ]

is syntax sugar for:

    push!!..! <e1> <e2> ... <en>        # with n exclamation marks

Similarly for the other statements, e.g. `set_pixel*`.

The arguments to `space` and `data<N>` must be compile time constants,
except that `data8` also accepts labels.
```

An assembly program consists of a sequence of statements. Each statement consists of an optional label (followed by a colon), an instruction mnemonic and some optional operands. If operands are present, a sequence of operators `!` needs to be added after the mnemonic, as many `!` as operands. The symbol `#` marks the beginning of a comment that ends at the end of the line. Multi-line comments are not supported.

The assembler provides an alternative to the native `push<N>` instructions through the unique `push` mnemonic followed by the assembly operator `!`. The assembler determines which native `push<N>` instruction fits better regarding the operand (if it is a constant value known at compiling time or not). The operator `!` can be used to push one operand before the execution of the accompanying instruction or to push two operands if used in the form `!!`. A list of values enclosed in square brackets can be pushed onto stack using the operator `*`.

Although the mnemonic `get_sp` is not allowed, the TOS can be represented through the operator `&`. In this way, expression `&0` represent the value of SP (the address of the TOS), `&1` is the address of the TOS plus 8 (SP+8), and so on. To refer to the content of such stack positions, operator `$` behaves in a similar way.

In order to allocate space for global variables with their corresponding initial values, use the directive `data<N>` followed by a list of values enclosed in square brackets (`[ ]`). The value of `N` as in `push<N>`, `load<N>` and `store<N>` can be 1, 2, 4 or 8 bytes:

```
here:                   # 'here' is a code label (or data label)
    push!  1            # push value 1
    push!! beep here    # 'beep' (address of the variable) is pushed
                        # before 'here' on the stack
    load8! beep         # 126 (content of the variable) is pushed
    exit                # program ends
beep:                   # 'beep' is the address of an 8-byte word
    data8  [126]        # (it can be a global or static variable)
```

The binary is generated as a *position independent code*, so that all labels (data references and jump destinations) are computed as an offset with respect to the current PC value. This way, the `get_pc`

instruction becomes unnecessary in the generated assembly. Some arithmetic expressions in prefix form are also supported as operands. These expressions can be recursively used:

```
start:
    load8! (+ beep 8)            # push the value -1 on TOS
    load8! (+ (load8 pointer) 8) # also push -1 (indirect access)
    exit            # program ends
beep:               # 'beep' is the address of a vector of 8-byte values
    data8 [126 -1]
pointer:
    data8 [beep]    # the value of the variable 'pointer' is the
                    # address of beep
```

The `ivm64-gcc` compiler outputs a subset of all of this syntax. Complex expressions are split into several statements in such way that each code statement represents a single action, that involves a mnemonic and one single operand (immediate, SP relative or PC relative (label relative)). Examples of these statements output by the compiler are these:

```
start:
    push!  130          # TOS is 130      stack: [130]
    push!  5            # TOS is 5        stack: [130 5]
    add                 # TOS is 135      stack: [135]
    add!   -1           # TOS is 134      stack: [134]
    load8! beep         # TOS is 126      stack: [134 126]
    load8! pointer
    load8               # indirect load   stack: [134 126 -1]
    load8! &0           # duplicate TOS   stack: [134 126 -1 -1]
    exit                # program ends
beep:                   # 'beep' is the address of a vector of 8-byte values
    data8  [126 -1]
pointer:
    data8 [(+ beep 8)] # the value of the variable 'pointer' is
                       # the address of beep plus 8
```

4.2.1 Restrictions to the assembler

The current linker implementation (`as/ld` scripts) imposes some restrictions to the assembler because files are processed line by line[2]. To guarantee that an assembly code will be linked without problems, restrictions shown in table 2 should be met.

4.3 Stackification scheme

One of the challenges found when porting the GNU C compiler to the IVM64 architecture has been the lack of registers. One the one hand, IVM64 is a stack machine with only two specific purpose registers, PC and SP. There are neither general purpose registers, nor Frame Pointer (FP). On the other hand, the GCC internal intermediate representation (RTL) makes use of symbolic registers (*pseudoregisters* in gcc terminology) that need to be mapped to physical (hard) registers during the crucial *register allocation* pass (see figure 1). The register allocator is able to *spill* pseudoregisters into the stack when not enough registers are available in the architecture, but to get reasonable results a bunch of registers is expected to be available.

This is why we need to pretend that there are registers in the target architecture, although there are none actually. With this aim, and following a approach similar to that in [8], a set of general purpose

[2]The `as/ld` scripts are based on common Unix tools like `grep`, `awk`, ..., that scan the input files line by line.

Restriction	Recommended	Not supported

One and only instruction per line	`push! 3`
	`push! 4`
	`push! 5`
		`push! 3 push! 4`
		
		`push!`
		`5`

Label/alias declarations: alone and in one only line	`main:`
	
	`var:`
	`data1 [0]`
		`main`
		`:`
		
		`var: data1 [0]`

Data declarations in one only line	`data1 [0 0 0 0]`	`data1 [0 0` `0 0]`

Table 2: Some restrictions to the assembler to be processed by `ld/as` scripts.

registers will be defined virtually, mapped onto the stack. Note that they are not global registers in the conventional sense, because these registers are in the stack, but registers that live only for the current function, similarly to local variables or arguments.

This way, for the GCC purposes, the IVM64 target is defined including the registers shown in table 3. Some remarks about these registers:

- For GCC, the program counter (PC) is an implicit register without register number.
- The FP register is used by the compiler until the register allocation, where it must be eliminated. Basically FP is expressed in function of SP, according to certain elimination rules set by the target description; it should never appear in the final assembly output.
- TR is an instrumental register that represents the TOS. Writing to it involves to push a word on the stack. Reading from it involves to pop 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 be handled only following certain rules.
- There are 16 general purpose registers (GPR)[3] named as AR, X1, X2, ... X15. The register AR is used by functions to return a 64-bit value (or smaller). When returning a 128-bit value, the pair (X1,AR) is used, being AR the less significant 64-bit word.

Register number	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,X2, ... X15	Other general purpose registers

Table 3: Registers defined for compiler purposes.

Figure 3 shows the stack layout allocated immediately after the call to a given function, including the stack mapped registers. Several regions can be identified: a deep stack associated to the stack frame of the caller, the arguments of the function which are passed on the stack, the return address, the stack frame which is automatically allocated by the compiler for local variables and spilled registers, and finally the stack mapped GPRs. Note that temporary values can be pushed for this function below this point. These may correspond to those operations with the special register TR.

[3]The choice of 16 general purpose registers is experimental, as no significant performance improvement were observed by increasing this number beyond 16.

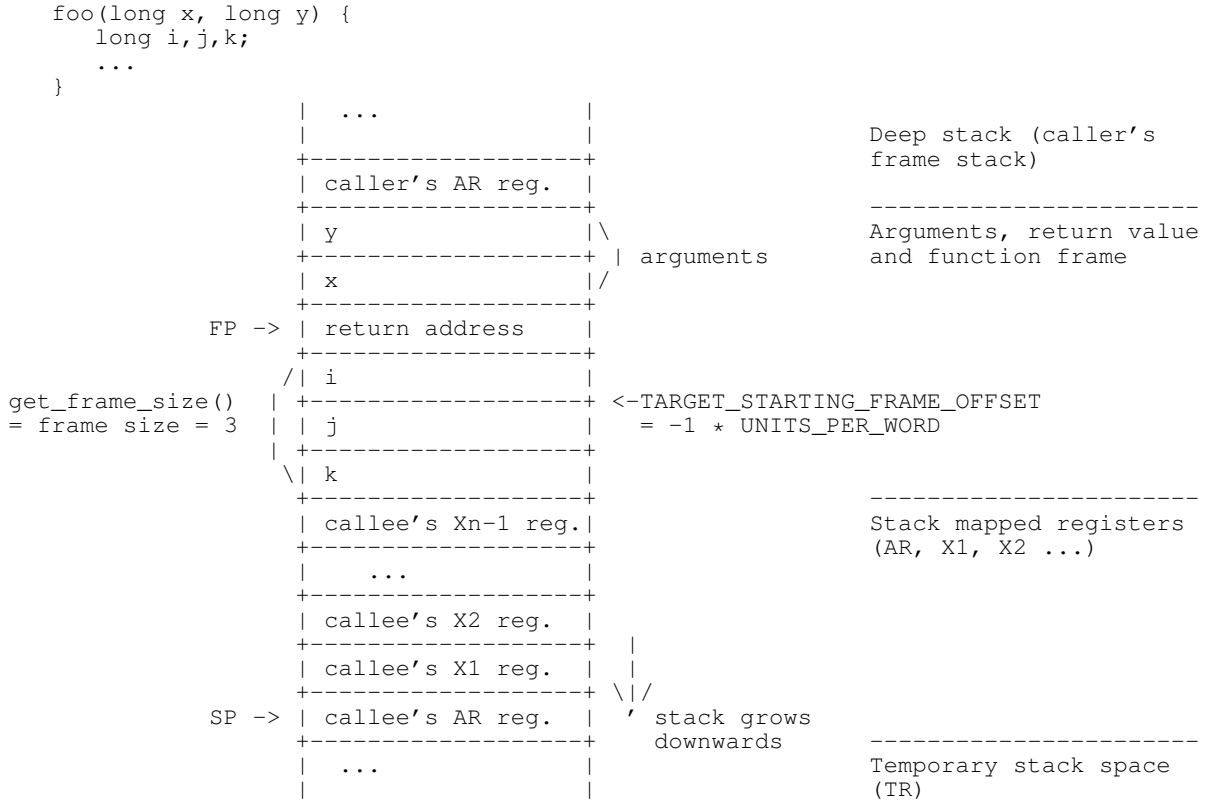

Figure 3: Example of the stack layout after the prologue of a given function. Several GCC macros of interest are displayed.

Observe that the target must track SP in order to locate the current position of the first GPR. In a function, SP can change by two ways:

- Explicit SP modification; for example when pushing an argument, SP is predecremented
- Operations on TR, in this case the gcc machinery has no knowledge about this SP modification

Several target variables are in charge of tracking SP:

- `ivm64_gpr_offset` which is updated in explicit SP modifications, registering the number of bytes by which SP is incremented;
- `ivm64_emitted_push` / `ivm64_emiited_pop` count the number of push/pop actions performed through the instrumental register TR.
- another variable `ivm64_stack_extra_offset` is used to fine tune stack imbalances while an RTL expression is output. This last variable must always be zero, before and after the RTL is printed.

4.4 Instruction expansion

During the expansion pass (see figure 1), the internal tree representation (GIMPLE) is expanded via a set of canonical RTL rules[4] that must be included in the machine description file (`ivm64.md`). It is during this phase when the instrumental register `TR`, that represents the TOS, is used to express those temporary operations performed on the top of the stack.

Observe that `TR` is an special register, and GCC has to be instructed not to use it in a general way. It must be avoided performing transformations that give rise to wrong programs, for example due to a stack imbalance. With this aim, some operations on `TR` has been defined in the machine description (`.md` file) using the RTL type UNSPEC (unspecific) that impose certain restrictions to the compiler when transforming these RTL expressions containing `TR`.

For example, consider the transfer `dst <- TR - TR`. What does it mean? Perhaps its meaning it is not `dst <- 0` as probably the GCC machinery would consider. Perhaps the developer's purpose is to express popping two operand from the stack, subtracting them and writing the result to its destination. Consequently the semantic of operating on `TR` needs to be specifically expressed in the machine description. The `TR` register should be used only for operations and transformations allowed in the `.md` file, but for nothing else. These RTL patterns define basic actions like push (`TR <- operand`), pop (`dst <- TR`), arithmetic operations (`TR <- TR op operand`), etc. Also there are other specific patterns, e.g. for peephole optimizations.

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

```
(define_expand "udivdi3"
[ (set (match_operand:DI 0 "general_operand" "")
       (udiv:DI (match_operand:DI 1 "general_operand" "")
                (match_operand:DI 2 "general_operand" "")))]
  ""
{
  ivm64_expand_push(operands[1], mode);
  emit_insn(gen_rtx_SET(TR_REG_RTX(mode),
                        gen_rtx_UDIV(mode,
                                     TR_REG_RTX(mode),
                                     operands[2])));
  ivm64_expand_pop(operands[0], mode);
  DONE;
})
```

Here functions `ivm64_expand_push` pushes an operand and `ivm64_expand_pop` pops the result. Once the result is popped, the content of the `TR` register is undefined. This fact must be communicated to the compiler because this register can not be read again at this point. For this reason the `ivm64_expand_pop` function needs to *clobber* it. It is in this way how the three-address SSA-based GIMPLE forms are translated into several one-operand instructions, with the help of the instrumental `TR` register.

[4]The list of canonical rules is large; there are rules that are mandatory to define for a target; others are optional to tune or optimize the code.

For this example, the RTL 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))
```

Notice that the emitted push and pop actions are expressed operating on TR, but actually not in a direct way but with RTL unspec_volatile constructions. As commented this prevent the compiler from doing incorrect transformations, given the special semantic of TR.

Finally, after all the compilation passes, the resulting RTL expressions must be printed as assembly instructions. Rules to output the assembly need also to be defined in the .md file. For example, next RTL pattern in ivm64.md describes how to write the assembly corresponding to the 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 are going to be written when found 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. Predicates are defined in predicates.md. Quoted letters represents *constraints* associated to addressing modes ("i" for immediate, "r" for register and "m" for memory). Target specific constraints are defined in file constrains.md.

4.5 Application Binary Interface (ABI)

The application binary interface (ABI) is a key piece for the interconnection between program modules written in C (compiled with ivm64-gcc) and hand written assembly code. Two important aspects of the ABI are the data layout (that can be shared among several modules) and the calling convention.

4.5.1 Data layout

Scalar 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-byte wide while the type double is 8-byte wide. IVM follows the *Little Endian* convention, so the least significant byte (LSB) of a word is placed on the lowest memory position. Two additional 128-bit types are supported by the compiler, that are mapped into 2 consecutive positions: long128 and complex double.

To reference a variable defined in C from an assembly code use the name of the variable (C compiler define the label) as it is defined. Use the appropriate size for the load or store instruction. Next examples correspond to the declaration and usage of global C variables:

```
# In C program      # Generated ASM          # Example of use in ASM code
char var1;              EXPORT var1          load1!   var1
                      var1:                  store2!  var2     # var2 = var1
                        data1 [0]
short var2;           ...
int var3;             ...                    store4!! 7 var3   # var3 = 7
long var4;              EXPORT var4          store8!! var4  &6 # save in TOS
                      var4:                                    # plus 6*8 the
                        data8 [0]                              # address &var4
```

A vector, or array, of N data elements (indexed from 0 to N-1) is a succession of consecutive memory positions where each element is placed on a position according to its index to allow pointer arithmetic as defined in standard C. The address of the vector matches the address of the first element of the vector (whose index is 0). There are no alignment between elements in an array nor padding:

```
# In C program      # Generated ASM          # Example of use in ASM code
long a[3];              EXPORT a             store8! 34 (+ a 8)  # a[1]=34;
                      a:
                        data8 [0 0 0]
```

Reciprocally, a data declared in ASM code can be used in a C program by declaring it as an external symbol:

```
# Generated ASM                          # Example of use in C
    EXPORT a                             extern long a[3];
a:                                       void foo () {
    data8 [1 2 3]                           a[1] = 6;
                                         }
```

In the case of structures, it should be noted that a padding can be added between structure members, or at the end of the aggregate type, but never at the beginning before the first member. A structure must always occupy an integer number of 64-bit words in memory, so a padding may be added at the end. On the other hand, a padding between member assures that each member is aligned with respect its data type size relative to the start of the structure. This example shows how structure members are padded:

```
struct s {                  S:
    char a;                     data1 [1]  # no padding at the beginning
    int b;                      data2 [0]  # padding: int type must start
    short c;                    data1 [0]  # in a multiple of sizeof(int)
    long d;
    char z[3];                  data4 [2]
} S = {1,                       data2 [3]
       2,                       data4 [0]  # padding: long type must start
       3,                       data2 [0]  # in a multiple of sizeof(long)
       4,
       {'a', 'b', 'c'}};        data8 [4]
                                data1 [97 98 99]
                                data4 [0]  # padding: at the end to get
                                data1 [0]  # a size multiple of 8 bytes
```

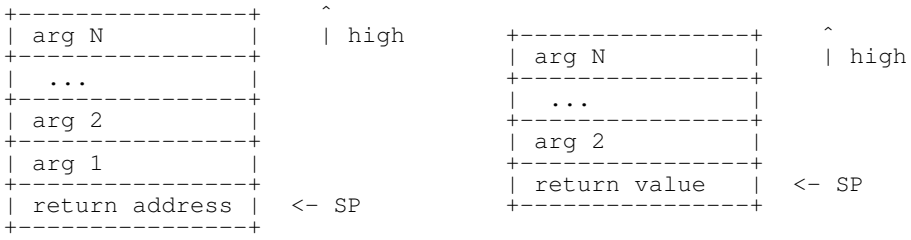

Figure 4: On the left, the stack content immediately after a call instruction; on the right the stack content just after the return instruction. A function with N arguments, as `f(arg1, arg2, ..., argN)`, is assumed.

4.5.2 Calling convention

This section describes the Application Binary Interface (ABI) related to the calling convention used by `ivm64-gcc`. Assembly codes aspiring to interoperate with gcc-generated functions must satisfy these conventions that defines two features: how arguments are passed to the function and how the return value is returned.

A unique ABI is used for all functions regardless the number of arguments (no arguments / fixed number of arguments / 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 (arguments are passed from right to left in the order of the C declaration).

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 (note that the return address has been just popped, so the return value is on the top of the stack).

   At this point, the caller copies the return value (e.g. to AR) and releases arguments, if necessary.

   Stack content after the call and return instructions is shown in figure 4.

   Note that the callee stores the return value in the stack position immediately above the return address, overwritting the first C argument (or the corresponding stack slot if there are no arguments).

An example of the assembly code generated for a C code calling a function is shown in figure 5. Some remarks about the calling convention follows:

- Stack grows downward. The macro `STACK_GROWS_DOWNWARD` is set to 1 in `ivm64.h` header file. This means that SP register must be decremented to make room on stack for the return value, the arguments and the return address. Arguments are pushed one by one.

- Every pushed argument needs to be a multiple of a word (64 bits). With this purpose, the macro `STACK_ALIGNMENT_NEEDED` is set to 1 in `ivm64.h`. If the size of an argument is larger than 8 bytes (e.g. structs) then the following arguments is placed in a position that is a relative multiple of 8. Native instruction `load<N>` and assembly mnemonic `push!` do this job properly.

- Arguments grow downward. The macro `ARGS_GROW_DOWNWARD` is defined as 1 in `ivm64.h` header file. This mean that arguments are pushed successively from right to left on the C function definition. The leftmost argument is pushed last.

```
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 registers

        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  # prolgoue

        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 5: Example of a function call (C code on the left; generated IVM64 assembly on the right)

- The return address is pushed (there is no link register) after the arguments. Once this address is pushed, the control is transferred to the *callee* function through a jump instruction; the assembly mnemonic `call` does both actions (pushing the return addres and jump to the function).
- The return value is returned on stack, overwriting arguments (if they exist) or on the previously allocated space (if return value size is larger than arguments size).
- There is no static chain (as there is no hardware FP register).

Calling a C function from ASM Here you can find some examples of how a function that was generated by `ivm64-gcc` can be called from an assembly code. The actions to be done by the caller are those described previously: (1) allocate space for the return value, (2) push arguments, (3) call the function, (4) store the value (if any) and (5) free the remainder argument left on the stack (if any).

Note that when pushing a memory operand, IVM64 instructions `load<N>` need to be used, but a constant should be pushed with `push!` instead. In next examples, we consider all passed arguments to be references to memory (labels).

Example of calling a C function returning a value, with more than one argument:

```
# calling a C function foo(a, b, c)
# valid prototypes may be  'long foo(int a, int b, int c)',
#   'long foo(long a, ...)', 'long foo(long a, long b, ...)'
push! 0   # allocate a stack slot for return value
load8! c  # push arguments
load8! b
load8! a
```

```
  call! foo
  store8! (+ &0 (* 3 8))        # caller copies the return value
  set_sp! (+ &0 (* (+ 3 -1) 8)  # caller releases the remaining arguments
  # now the return value is on the top of stack
```

Example of calling a C function returning a value, with no arguments:

```
  # calling a C function with prototype 'long foo()', as foo()
  push! 0   # allocate a stack slot for return value
  call! foo
  # copying the return value is not necessary and there are no arguments to release
  # now the return value is on the top of stack
```

Example of calling a C function returning a value, with only one argument:

```
  # calling a C function with prototype 'long foo(long a)', as foo(a)
  push! 0   # allocate a stack slot for return value
  load8! a  # push only one argument
  call! foo
  store8! &1         # caller copies the return value
  # there are no arguments left to release
  # now the return value is on the top of stack
```

Example of calling a C void function, with several arguments:

```
  # calling a void C function foo(a, b, c)
  # prototype 'void foo(int a, int b, int c)'
  load8! c  # push arguments
  load8! b
  load8! a
  call! foo
  # there is no return value
  set_sp! (+ &0 (* 3 8))  # caller must release all the arguments
```

Calling an ASM function from C The next example shows the structure of an assembly function that need to be called by a C function , that is, the *caller* is written in C and compiled with `ivm64-gcc`, whereas the *callee* is written in assembly.

Suppose that the ASM function to be called is declared in the C caller code is the following one, in charge of computing $a + b \cdot c$:

```
    uint64_t foo(uint64_t a, uint64_t b, uint64_t c);
```

The ASM code fitting this declaration could be:

```
    EXPORT  foo
foo:
    # compute a+b*c
    load8! &3  # push argument c
    load8! &3  # push argument b (be aware that one value was pushed)
    mult
    load8! &2  # push argument a (be aware that b*c is on the stack)
    add
    store8! &2 # pop out the result value over the first argument
    return
```

Using a more compact syntax:

```
    EXPORT  foo
foo:
    push!       0       # push0, make room for a local var on stack
    store8!!    (* (load8 &3) (load8 &4)) &0    # store the product
    store8!!    (+ (load8 &0) (load8 &2)) &0    # store the sum
    store8!     &2      # pop out the result value over argument 1
    return
```

Or even more compact:

```
    EXPORT foo
foo:           #      a2 -> &2   a3 -> &3    a1 -> &1   return -> &1
    store8!!   (+ (* (load8 &2) (load8 &3)) (load8 &1)) &1
    return
```

Passing and returning structures Structures and unions are passed similarly as single type arguments, by being copied on the stack. As they are padded to be multiple of 64 bits, there is no alignment mismatch with the rest of the arguments. For not small structures[5], GCC usually invokes string functions like `memcpy()` or `memset()` to copy them on the stack.

The mechanism used by GCC to return a structure is controlled by the target macro `DEFAULT_PCC_STRUCT_RETURN` which is enabled for IVM64. In this way, the caller is made responsible for reserving place for the returning aggregate type on the frame stack. A pointer to this reserved memory area is passed as a hidden argument that is pushed first.

4.6 Inline assembly

Assembly code can be embedded directly in the C program by means of the GCC syntax for inline ASM, which corresponds to this template:

```
asm volatile( "ASM code"
          : OutputOperands
          : InputOperands
          : Clobbers
          : GotoLabels)
```

Input and output operands are expressed as a comma separated list of items of the form `"<constraint>"(operand)`. A constraint consists of a letter referring to a possible addressing mode. In this implementation constraints can be: `"i"` for immediate operands, `"m"` for variables in memory, and `"r"` for registers. With immediate operands only constants can be used (e.g. `"i"(3)`). If the `"r"` constraint is used as input, any generic expression is stored in one stack mapped GPR before being used in the assembly code. Memory operands (`"m"`) can refer to global or static variables as well as stack variables like arguments or locals. Output operands are restricted to the memory constraint, and because they are written, an equal sign must precede the letter: `"=m"`. In the string containing the ASM code, operands are referenced as `%0, %1`, ... where the number is the position of the operand in its appearance order in the operand list.

When combining C and assembly codes, it is recommended to make use of inline assembly when possible. A common scenario is when designing IVM64 input/output routines that needs to include specific IVM64 I/O instructions. This way, the inline ASM code can reference any C variable (local

[5]Those bigger than one 64-bit word.

or global, arguments, ...), avoiding the need to write separate assembly modules and handling explicitly details as the calling convention. Some examples illustrating the inline assembly follow.

This is an example of a function printing a character. In this case the operand is in memory which is read (input operand in the the template):

```
#ifdef __ivm64__
static int outbyte(unsigned char arg) {
    int retval=arg;
    asm inline ("load1! %0" :: "m" (arg));
    asm inline ("put_char");
    return retval;
}
#define putchar(c)  outbyte(c)
#endif
```

In these two examples assembly instructions read function arguments. Observe that the compact (sugared) assembly notation allows the programmer to ignore the stack offset variations due to push/pop operations:

```
void ivm64_new_frame(long width, long height, long rate)
{
      asm volatile("new_frame* [(load8 %0) (load8 %1) (load8 %2)]"
                   :: "m"(width), "m"(height), "m"(rate));
}

void ivm64_set_pixel(long x, long y, long r, long g, long b)
{
      asm volatile("set_pixel* [(load8 %0) (load8 %1)"
                   "             (load8 %2) (load8 %3) (load8 %4)]"
                   :: "m"(x), "m"(y), "m"(r), "m"(g), "m"(b));
}
```

In next examples assembly instructions must write some stack variables. The stack imbalance caused when pushing values must be taken into account when writing to variables placed on the stack:

```
void ivm64_read_frame(long* widthp, long* heightp, long frameindex)
{
  // these local variables are in the stack
  long width = 0, height = 0;

  // Push 16 bytes
  asm volatile("read_frame! (load8 %0)" :: "m" (frameindex));
  // Pop 8 bytes
  asm volatile("store8! (+ 16 %0)" : "=m" (height));
  // Pop 8 bytes
  asm volatile("store8! (+  8 %0)" : "=m" (width));

  *widthp = width;
  *heightp = height;
}

long ivm64_read_pixel(long x, long y)
{
  // This local variable is on the stack
  long v = 0;

  // Push 8 bytes
  asm volatile("read_pixel* [(load8 %[x]) (load8 %[y])]"
               :: [x] "m" (x), [y] "m" (y) );
  // Pop 8 bytes
  asm volatile("store8! (+ 8 %[v])" : [v] "=m" (v));

  return v;
}
```

Observe that if the temporary variables are defined as static, they are not placed on the stack and can be referenced without taking into account the stack imbalance:

```
void ivm64_read_frame(long* widthp, long* heightp, long frameindex)
{
  // these variables are NOT in the stack
  static long width = 0, height = 0;

  // Push 16 bytes
  asm volatile("read_frame! (load8 %0)" : : "m" (frameindex));
  // Pop 8 bytes
  asm volatile("store8! %0" : "=m" (height));
  // Pop 8 bytes
  asm volatile("store8! %0" : "=m" (width));

  *widthp = width;
  *heightp = height;
}

long ivm64_read_pixel(long x, long y)
{
  // This static variable is NOT on the stack
  static long v = 0;

  // Push 8 bytes
  asm volatile("read_pixel* [(load8 %[x]) (load8 %[y])]"
               :: [x] "m" (x), [y] "m" (y) );
  // Pop 8 bytes
  asm volatile("store8! %[v]" : [v] "=m" (v));

  return v;
}
```

Next example provides a pointer to the start of an area allocated with directive `space`, that could be used to implement a `malloc()` function. In this example, the stack imbalance is managed with an auxiliary array; observe the use of the C labels for jumping as a data reference:

```
void *get_heap_pointer(){
    // Auxiliary  array to compensate stack imbalance
    void *aa[1];

    asm volatile goto ("jump! %0" :::: malloc_skip);

malloc_space:
    asm volatile ("space %0" :: "i"(IVM64_MAX_HEAP_BYTES));

malloc_skip:
    // push 8 bytes
    asm volatile goto ("load8! %l0"    :::: malloc_space);

    // 8 bytes were already pushed
    asm volatile ("store8! %0" : "=m"(aa[1]));

    void *heap_start_p = aa[0];
    return heap_start_p;
}
```

4.7 Compiler tool-chain

One of the goals when designing the IVM compiler has been to keep things as conventional as possible, in such a way existing C projects could be ported and built with minimum effort[6]. In the standard compilation flow (see figure 6), `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`). These are the steps followed:

- First, `cpp` translates the input C source file (i.e., `prog.c`) into and 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.

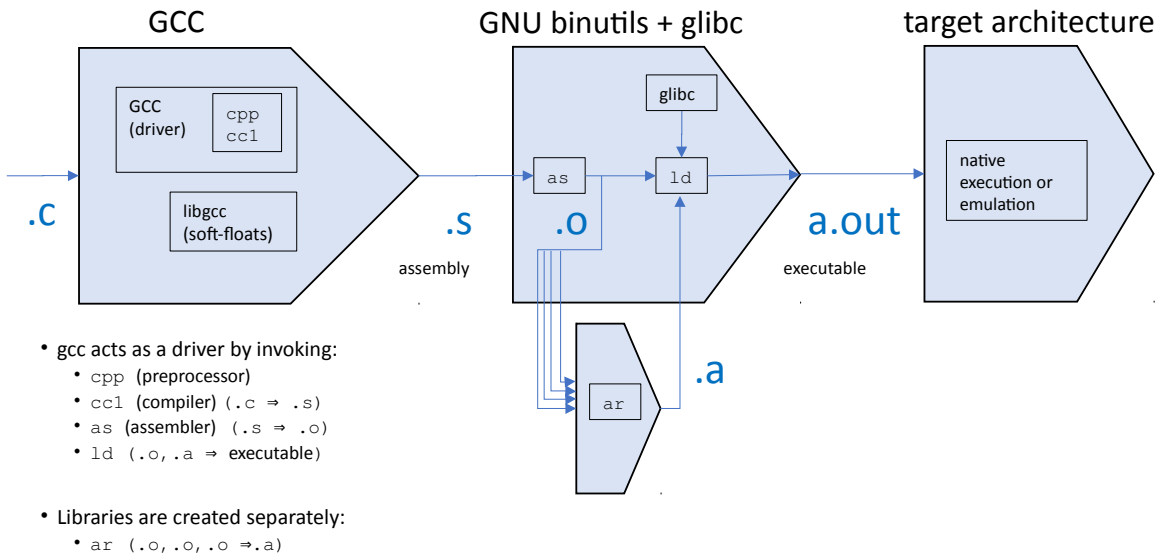

Figure 6: GCC tool-chain.

Nevertheless no linkable binary object format was defined along the development of this project, so that this standard compilation flow needed to be adapted. This led to the development of a full tool-chain that works completely in assembly format, relying the binary generation to a external tool, after the linking. With this purpose it was introduced the following files types:

- An *assembly object* is the result of applying 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[7]. The aim of this renaming process is to assure that local symbols in one object will not collide with other local symbols (with the same name) in another object when combining several objects together during the linking. This suffix serves as an unique object identifier.
- An *assembly executable* is the result of combining multiple *assembly objects*, including those coming from libraries. This linking process is carried out by the `ld` program.

Note that the assembly object format is not strictly necessary. The plain assembly would be enough if `ld` is in charge of processing colliding local symbols when concatenating all assembly files. Nevertheless the assembly objects are introduced for a better performance. If the disambiguation of local symbols is

[6]For example, consider the effort of adapting configuration scripts, makefiles, etc.
[7]Local symbols refer to labels or abbreviation that are not declared as global with the EXPORT clause.

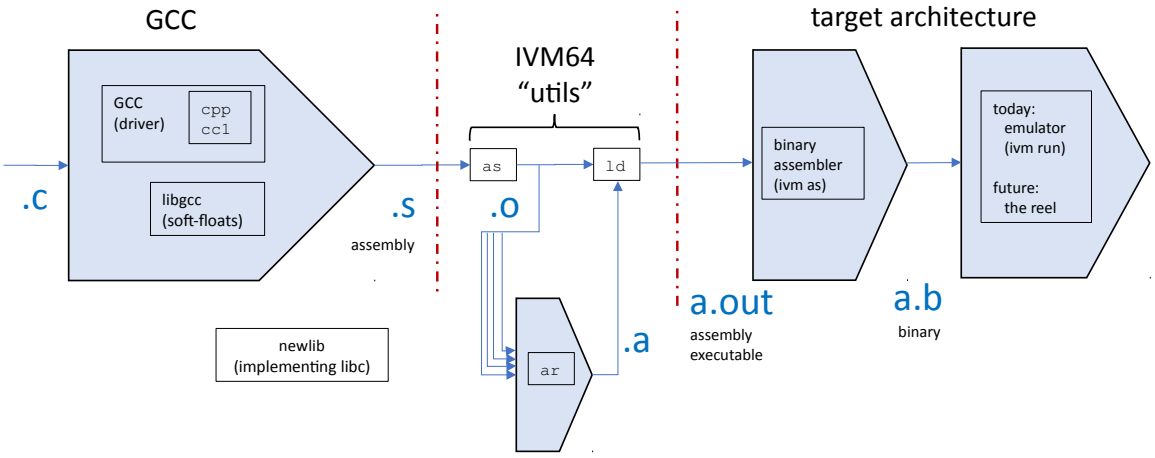

Figure 7: GCC IVM64 tool-chain.

done on creating the object, `ld` can save this processing. And this is specially interesting when linking with libraries, as libraries are basically a container of many already precompiled objects. Even so, `ld` always re-check for colliding local symbols.

Figure 7 shows the IVM64 compilation flow. Programs `as` and `ld` have been implemented as shell scripts. The `ivm64-gcc` driver expects certain options when invoking the assembler and linker. Consequently, scripts `as` and `ld` can be called with a minimal command line options in order to be compatible with the gcc driver. These options are:

- Script `as` needs the output object and the input assembly file names. Both arguments are mandatory. The last argument is the name of the assembly input file. For example:

  ```
  as -o object.o file.s
  ```

- Script `ld` has to specify the objects to link and libraries (if any). Libraries are searched in paths specified with `-L`. Libraries to be linked are specified with flag `-l`. There can be several `-L` and `-l` options to specify different library search directories and/or libraries, respectively. Flag `-lname` denotes library `libname.a` or `libname.so`. If no `-o` option, the output file name is `a.out` by default. With the optional flag `-e`, the name of a global label can be used as the program entry point. An example of invoking `ld` would be:

  ```
  ld [-o outfile] [-L libdir] [-lname] [-e entry] obj1.o obj2.o ...
  ```

An additional feature of the `ld` script is the ability of creating a true executable by means of a *shebang* header added to the final assembly output. This is done when the output file extension is none of those shown in table 4, that have an specific meaning. This feature is specially useful for testing the compiler output in today's platforms, where the compiler was compiled. This way, the final output of `ivm64-gcc`, the *assembly executable*, is just a concatenation of:

- A shebang header (if the extension is not specific), that enables its execution,
- the `crt0.o` startup file, which is placed at the very beginning,
- all object files of sources being compiled (which are really assembly files),
- the standard C libraries, and other libraries provided in the command line.

Two classes of libraries, static (`.a`) and dynamic (`.so`) are supported[8]. Libraries with extension `.a` can be created with the standard `ar` utility, e.g. `ar cr obj1.o obj2.o ... -o libmy.a`. Libraries with extension `.so` are basically a concatenation of assembly objects that can be created with the

[8]The terms *static* and *dynamic* refer to the format of the library rather than how they are loaded. The whole IVM64 compilation process is static, that is, all used functions are included in the final binary, as no dynamic loader exists.

Extension	Used for
`.s`	Assembly (as generated by: `ivm64-gcc -S`)
`.S`	Assembly (special assembly files like handwritten assembly, initialization, ...)
`.o`	Assembly object (as generated by: `ivm64-gcc -c`)
`.a`	Static library (as generated by: `ar cr obj1.o obj2.o ...`)
`.so`	Dynamic library (as generated by: `ivm64-gcc -shared obj1.o obj2.o, ...`)

Table 4: File extension convention.

same driver (`ivm64-gcc -shared obj1.o obj2.o ... -o libmy.so`). Script `ld` 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. This *dead code elimination* is performed by default, but it can be disabled, if desired, to make the linking faster (at the expense of a larger output). To disable such a feature, `ld` can be invoked with the flag `-mno-dce`. When calling the driver this can be done this way: `ivm64-gcc -Xlinker -mno-dce ...`

Another remarkable feature of the linker script is its ability to *relink* the output with new sources, because in case of global symbol repetition, `ld` discards all the occurrences except the first one. *Relinking* is not specially efficient from the binary size viewpoint, since the libraries may be included several times, but it is a powerful feature for testing (see section 5). For example consider one of the codes from the *ivm-apps* suite [14], `tiff2ps`. After building the package, the *assembly executable* `tiff2ps` is produced. But we cannot test it because it needs a file system. We can generate a folderless filesystem with [15], including some test input files. Let `ivm-fs.c` be the source code with the generated filesystem. Now we can relink the output with this source:

```
ivm64-gcc ivm-fs.c tiff2ps -o tiff2ps-fs
```

The new generated executable `tiff2ps-fs` can now be tested with the files included in the filesystem, for example `tiff2ps-fs input.tiff`.

4.8 External tools

In order to complete the full tool chain, two external tools are necessary (see figure 7):

- The assembler, that provides the binary translation of the `ivm64-gcc` output. It is the so-called *ivm implementation* [13] which is in charge of doing this translation.

  For generating the binary , the ivm implementation can be invoked as:

  ```
  ivm64-gcc program.c  # generates the assembly executable a.out
  ivm as a.out         # generates the binary image a.b
  ```

- An emulator of the IVM64 machine, that allows executing the resulting binary in today's computers. Note that the emulator is only necessary to executing the codes in today's computers, mainly for testing, as the final purpose of the generated binary is to be stored in the reel for future developers.

  In the course of this project, several emulators has been developed:

  - The ivm implementation itself, that can simulate the binary (invoked as `ivm run a.b`)
  - A plain emulator written in C provided together the ivm implementation (`vm.c`)

- *Yet another (fast) ivm emulator* [16], an optimized emulator written in C
- A port of QEMU [17] for the IVM64 architecture [18].

Any emulator is expected to have a common set of options in order to be compatible between them and to make works the *shebang* included in the *assembly executable* (see section 4.7). Table 5 summarizes these options. As a convention, the emulator writes its messages to `stdout`, while output instructions printing to the terminal (like `put_char`) write to `stderr`.

Flag	Meaning
`-m <bytes>`	Virtual machine memory size in bytes
`-a <file>`	Specify an argument file
`-e <entry>`	Set a global (exported) label as the program *entry point*
`-i <in_dir>`	Directory for input instructions
`-o <out_dir>`	Directory for output instructions

Table 5: Options supported by an IVM64 emulator (its use may be optional).

Assembly executable files generated by `ivm64-gcc` offer the valuable possibility of executing the resulting file as it was a true binary in the host computer[9]. This is performed thanks to a *shebang* sequence that acts as an interpreter in charge of calling the assembler (`ivm as`) and the emulator. This way, a C program can be compiled and executed as:

```
ivm64-gcc program.c  # The (assembly) executable a.out is generated
./a.out hello        # Invoke it as a standard executable with arguments
```

The header in the assembly executable: (1) generates the binary file, (2) prepares program arguments getting them from `/proc/$PPID/cmdline`[10], (3) invokes the emulator, injecting program arguments through the argument file to be processed by `crt0.o`, (4) returns the value generated by the program (last value pushed on the stack). All the program output is redirected to `stdout` whereas other messages are redirected to `stderr`.

When invoking an executable it is possible to specify which emulator to use and how to deal with the memory size and the input/output directory if needed. To do so, use these environment variables `IVM_EMU`, `IVM_MAXMEM`, `IVM_OUTDIR` and `IVM_INDIR`:

If defined `IVM_EMU` to be a valid simulator, it is used as simulator, otherwise it simulates with 'ivm run'

If defined `IVM_MAXMEM=<N>`, it is used as memory size (passed to the emulator with option `-m <N>`), otherwise it uses 64MB

If defined `IVM_OUTDIR=<dir>`, and directory `<dir>` exists, it is passed to the emulator with option `-o <dir>`

If defined `IVM_INDIR=<dir>`, and directory `<dir>` exists, it is passed to the emulator with option `-i <dir>`

For example:

```
export IVM_MAXMEM=2000000000
export IVM_EMU=/path/to/ivm_emu_fast
./a.out
```

[9] A Unix/Unix-like operating system is assumed.
[10] The `/proc` filesystem is required

or in one line:

```
IVM_MAXMEM=2000000000 IVM_EMU=/path/to/ivm_emu_fast ./a.out
```

It the program includes IVM64 I/O instructions (e.g. `put_char`, `new_frame`,...), the input and output directory can be specified (do not forget creating these directories prior to the execution):

```
IVM_INDIR=/tmp/in IVM_OUTDIR=/tmp/out IVM_EMU=/path/to/ivm_emu_fast ./a.out
```

4.9 Libraries

The GCC compiler needs to be accompanied by the standard C libraries in order to produce fully functional codes. Two important libraries has been ported to the IVM64 architecture:

- The *libgcc* library, which includes a collection of routines for floating point emulation (`libgcc.a`). This library takes part of the GCC distribution. Remember that the IVM64 architecture supports only integer arithmetic.
- The *newlib* library [19, 20], which provides the C standard library. It is distributed as two separates sources: libgloss and the newlib itself. The first one implements a set of primitives (similar to elementary *system calls*) which newlib is build upon. Both newlib and libgloss has been integrated into the sources of GCC. After building newlib and libgloss the following files are generated:
  - `crt0.o`: the C runtime startup file
  - `libc.a`: the standard C library
  - `libm.a`: the mathematical C library

By default, `crt0.o` is linked as the first object, and libraries `libgcc.a` and `libc.a` are linked to the executable by the driver[11]; thus, the two next invocations are equivalent:

```
ivm64-gcc 00-main.c           # by default adding -lc -lgcc
ivm64-gcc 00-main.c -lc -lgcc # thus, this equivalent to the command above
```

Nevertheless, as usual, `libm.a` must be explicitly set for those programs using mathematical functions (sin, cos, pow, ...): `ivm64-gcc program.c -lm`

Some driver flags are available to suppress the liking of the startup file and libraries when they are not needed: `-nostartfiles` for not using crt0, `-nodefaultlibs` for not using libc and libgcc, `-nostdlib` for not using neither crt0, nor libc, nor ligcc. In case the crt0 file is not linked, you must provided how to start the program. There are two choices: define an explicit entry point with flag `-e`, or using a special `ld` option, `-mcrt0`, that includes a minimal startup code at the beginning of the assembly that calls the function `main()`[12]. Some examples of theses flags follows:

```
# no libc, no libgcc, no crt0, defining entry point
ivm64-gcc prog.c -nostdlib -e main

# no crt0, minimal startup added by ld
ivm64-gcc prog.c -nostartfiles -Xlinker -mcrt0

# no libc, no libgcc, using mylib.a instead
ivm64-gcc prog.c -nodefaultlibs -lmy
```

[11] That is, `ivm64-gcc prog1.c prog2.c ...` will call the linker as `ld crt0.o prog1.o prog2.o libc.a libgcc.a`

[12] In particular, an IVM64 assembly sequence as: `push* [0 0 0] call! main exit`

4.9.1 Startup file `crt0`

The C run-time startup file is linked by default as the first object of the program, unless a specific entry point is defined (`-e` flag). The IVM64 `crt0` provided with newlib is responsible for:

- providing a default start point through the global label `_start`,
- initializing the global variable `__IVM64_heap_pointer__` pointing to the beginning of the heap area[13],
- initializing the arguments of the function main, `argc` and `argv`,
- calling the function `main(int argc, char* argv[])`,
- on exit, leaving the `main()` return value on the stack.

The design of `crt0` is dependent on how the *ivm implementation* builds the binary. In figure 8, it is shown the memory layout after loading the binary as well as how the program is invoked. Observe that an initialization code is placed before the program itself. This initialization code needs to invoke the program, and this is done differently if an entry point was specified or not. Located after the program, two memory areas follow: the argument file and memory allocated through `SPACE` clauses in the assembly. From this point the free memory starts (not taking part of the binary): the heap after the space area, and the stack starting at the highest memory position. In case no entry point is defined, a jump to the first program position is done; and two words before the first program position store the pointers to the beginning of the argument and heap area respectively. In case an entry point is defined, the initialization code calls the entry point as a function, passing three arguments; the first one is the pointer to the heap start.

As commented in section 4.8, the `ivm64-gcc` output can be launched in a Unix shell as a true executable with arguments. Arguments are taken from the file `/proc/$PPID/cmdline`, which contains the list of arguments separated with the `'\0'` character. It is just this file which is used as argument file[14]. The `crt0` is in charge of parsing the argument file in memory, creating the `argv` structured which is passed to function `main(int argc, char argv[])` along with the number of arguments (`argc`).

It is worth mentioning that there is an interdependence between `ctr0` and some functions of newlib. For example, `crt0` initializes the pointer to the heap start[15], which is necessary for implementing dynamic memory allocation (`sbrk()` function). On the other hand, `crt0` invokes `exit()` that is a function included in libc.

4.9.2 Function `alloca`

As the IVM64 architecture lacks a frame pointer, a software implementation of `alloca()` is provided, included in the libc library. It is based on a portable public-domain implementation [21]. Function alloca can be used directly or wiht the builtin function `__builtin_alloca`.

Also it is invoked automatically when declaring variable length arrays (VLA). The alloca implementation has been supplemented with two intrisics `__builtin_stack_save` and `__builtin_stack_restore`, which are used to save storage space when VLAs are reused into structures like loops.

[13] This is necessary for function *sbrk()* in libgloss, called by *malloc()*

[14] Using the emulator flag `-a`. The argument file is placed in the argument file area depicted in figure 8.

[15] To relax this dependence, a preinitialization of the heap start pointer is done by `ld`, by adding a `space` directive at the very end of the program, getting from that the start of the heap area.

Figure 8: IVM64 binary layout and start up process for *ivm implementation* v0.37.

4.9.3 Functions `setjmp/longjmp`

Functions setjmp/longjmp are provided in the libc library. They can be invoked directly or via their builtin versions `__builtin_setjmp` and `__builtin_longjmp`. The latter two are directly expanded as assembly code. The jump buffer has been designed in order to be interchangeable between the library and builtin versions.

4.9.4 Optimized string functions

Most of string functions provided by newlib are optimized by aligning strings according with their size. Nevertheless, as IVM64 assembly has no support for alignment, codes will not take advantage of this feature, even they can be suboptimal. This is why these functions has been adapted considering the lack of alignment directives.

4.10 Reserved symbols

An important preprocessor macro declared by the compiler is `__ivm64__` which is defined as true. This macro is very useful when writing a C code that are to be compiled for both IVM64 and other platforms (e.g., x86). In these cases, some parts of the code may need to be conditionally compiled according to the target, for example those parts related with I/O.

Additionally, some symbols are reserved for the internal use of the compiler and libraries. Users should avoid redefining or using them without care in their C or assembly codes. These symbols are (in glob notation):

- `.L*`: symbols starting with `.L` are always treated by `as/ld` as local symbols; they are generated by the compiler for local labels.
- `__IVM64_*__`: some global (exported) variables used by libraries have this pattern, such as `__IVM64_heap_pointer__`, `__IVM64_exit_jb__` and `__IVM64_exit_val__`.
- `.X*`: some internal global symbols used by the compiler may start with preffix `.X`, e.g. `.XSC` for the static chain register.
- `__star` and `_start`: these two labels are defined by the `crt0.c` file. The label `__start` corresponds to the first position of the compiled program, which is the default entry point. Function `_start()` is the C run time start up routine.

4.11 Optimizations

The assembly code generated by `ivm64-gcc` benefits from three kinds of optimizations: (1) the optimization passes provided by GCC; (2) specific peepholes for this target; and (3) some lower level optimizations at assembly level.

Optimizations provided by GCC may include hundreds of passes[16], counting both the tree (GIMPLE) and RTL phases. According to the optimization level selected during the compilation (`-O<level>` flag), some of the passes can be enabled or disabled. These optimizations can also be fine tuned through many command line flags. By default the `ivm64-gcc` compiler has been configured to use the `-O2` level, which offers a good balance between speed (executed instructions) and size. It must be mentioned that some of the optimizations were too aggressive for this target, specially with the treatment of the `TR` register. The special meaning of this instrumental register makes some transformations not applicable. Nevertheless the number of disabled optimizations is very small, and has a very low impact in the overall performance.

The peephole transformations are a key element of optimization. Peepholes are applied as another GCC optimization pass, but they are target dependent. For IVM64, a wide set of peephole transformations are defined in file `peephole.md`. These transformations correspond to many common patterns found in the codes of interest, tested during the development of the compiler. There are two group of peepholes, those that express RTL transformations and those that specify assembly transformations. The RTL peephole phase occurs certain passes after the reload pass. The assembly peepholes are used at the very end, just when the assembly is printed.

Finally, a set of assembly level optimizations has been considered in the output phase. Among others these optimizations include: expressing some actions with the lowest possible number of native instructions, such as conditional jumps, sign/zero extensions and others; allocating on the stack only those RPGs used by a function not the full set of them; and, avoiding unnecessary load/store instructions when working with the `AR` register when this one is the TOS because `TR` is not in use.

[16] Passes can be listed in this way: `ivm64-gcc -fdump-passes file.c`

4.12 Using the compiler

The number of GCC options for fine-tuning the compilation process is very extensive. The user is addressed to [22] in order to explore options of interest or get more insight on them. The IVM64 cross compiler, `ivm64-gcc`, can be invoked with the most usual options of GCC. Some common options are shown in table 6. Others more specific have been commented in previous sections of this document. Note that without any of the options `-E`, `-S` or `-c`, the driver will invoke the linker, combining all the assembly objects in a unique output.

Some of the gcc options makes no sense for the IVM64 target and are ignored if specified. For example `-fno-omit-frame-pointer`, to enable the use of FP, because IVM64 lacks such a register. Another example is option `-g`, to enable debugging information, as no debugging support is currently implemented.

There is a set of options for developers, that allows debugging the compiler itself. For example, an important information while developing the target is to know the internal representation after each pass, which can be dumped with `-da`. Also several debugging annotations can be added to the assembly output with options like `-fverbose-asm`, `-dp` or `-dP`.

Option	Meaning
`-o file`	specify output file
`-E`	preprocess only (`cpp`)
`-S`	generate assembly only (`cpp + cc1`)
`-c`	generate object only (`cpp + cc1 + as`)
`-O<level>`	Select an optimization level (`level` in {`0, 1, 2, 3, s, fast`})
`-v`	verbose information
`-I<dir>`	add the directory `<dir>` to the list of directories to be searched for headers during preprocessing
`-L<dir>`	add the directory `<dir>` to the list of directories to be searched for `-l`
`-l<X>`	passed to `ld` to link with library `lib<X>.a` or `lib<X>.so`
`-Xlinker <opt>`	pass option `<opt>` to the linker

Table 6: Some common GCC options.

4.13 Structure of the target

The IVM64 back end has been developed for GCC version 10.2.0. By building it, the full C crosscompiler for the IVM64 architecture, `ivm64-gcc`, is generated [23]. The target comprised a set of sources files that are summarized in tables 7 and 8. Paths are relative to the GCC source distribution.

Every target needs a name to identify it. It is used, for example, to name some target dependent directories, where files describing the back end are placed, or to reference the target in other files. The name of the IVM64 target is *ivm64*.

First of all, the target must be registered in order to instruct GCC that a new target has been added. This is done in the GCC configuration files `config.sub` and `gcc/config.gcc`. The main target directory is `gcc/config/ivm64`. Under it, three key files with the core of the target description can be found:

- `ivm64.md`: this is the *machine description*; it holds the patterns for expanding and printing the RTL expressions into which the GIMPLE syntax tree is converted
- `ivm64.h`: here many features of the target are parameterized through the definition of a set of GCC macros

Main target files

Folder	Files
gcc/config/ivm64/	ivm64.c, ivm64.h, ivm64.md, as, ld, t-ivm64, peephole.md, constraints.md, predicates.md, ivm64-modes.def, ivm64.opt, ivm64-opts.h, ivm64-protos.h, README.txt
gcc/common/config/ivm64/	ivm64-common.c

Libraries

Folder	Files
libgcc/config/ivm64/	sfp-machine.h
libgloss/ivm64/	chown.c, close.c, configure, configure.in, crt0.c, execve.c, _exit.c fork.c, fstat.c, getpid.c, gettod.c, isatty.c, kill.c, link.c, lseek.c, Makefile.in, open.c, outbyte.h, posix.c, read.c, readlink.c, sbrk.c, stat.c, symlink.c, times.c, unlink.c, wait.c, write.c
newlib/libc/machine/ivm64/	aclocal.m4, alloca.c, configure, configure.in, Makefile.am, Makefile.in, setjmp.c, string/*.c
newlib/libm/machine/ivm64/	aclocal.m4, configure, configure.in, Makefile.am, Makefile.in
newlib/libc/misc/	ffs.c

Table 7: IVM64 target dependent files.

- `ivm64.c`: this includes the so-called *hooks*, that are target dependent functions required by GCC; also other functions supporting the description are included in this file.

Besides these three files, several others are placed in the target main directory. The `as` and `ld` scripts are two of them. They are going to be copied to its destination when installing the compiler, following the makefile rules in `t-ivm64`. File `peephole.md` contains the peephole definitions, which is actually included in `ivm64.md`. Other machine descriptions files included in it are `predicated.md` and `constrains.md`, that define target dependent predicates and constrains used when defining the target RTL patterns. The reader is addressed to [3] and the IVM64 target sources for a detailed understanding of the RTL patterns, and GCC macros and hooks that shape the target.

Some GCC files needed minor adaptations for aspects that are not possible to control with GCC macros or hooks. One example is how to tell GCC that function `alloca` is handled as a library function instead of a built-in. Table 8 shows the GCC files that required to be modified.

Note that in addition of the target description, used to build the `cc1` program for the given target, libraries need to be configured and some target dependent aspects to be defined. Tables 7 and 8 show those files requiring target customization for libraries libgcc and newlib (libc, libm, libgloss).

Finally it is worth commenting that instructions to build and install the compiler can be found in this file `gcc/config/ivm64/README.txt`.

5 Validation

The `ivm64-gcc` compiler has been thoroughly tested with an ample set of C programs. This set has included some generic benchmarks and also some specific applications related with the iVM project. The general tests have included:

- 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 perfomed), including the so called torture tests, specially designed to stress the compiler,

Configuration
config.sub
gcc/config.gcc
libgcc/config.host
libgloss/libnosys/configure.in
libgloss/configure.in
newlib/configure.host
newlib/configure.in
newlib/libc/machine/configure
newlib/libc/machine/configure.in
newlib/libc/sys/configure.in
Patched GCC files
gcc/opts.c
gcc/builtins.c
gcc/explow.c
gcc/tree-cfg.c
gcc/calls.c
gcc/passes.c
libgcc/unwind-sjlj.c

Table 8: Other GCC files that need modifications.

- The TCC testsuite [24] that contains nearly one hundred codes testing basic features of the C language,
- A collection of benchmarks from the LLVM testsuite [25]: *Olden, Prtdist, VersaBench, Fhourstones, lemon, llubenchmark, mafft, nbench, oggenc, spiff, viterbi, SMG2000, McGill, Shootout, Stanford*, and *CoyoteBench*,
- The two IVM64 emulators written in C, `vm.c` from [13] and the fast emulator [16], were also compiled for the IVM64 architecture. Emulated emulations were carried out successfully.

And the specific iVM applications has been:

- A static version of the *boxing* code [26] in charge of decoding (*unboxing*) the reel data frames,
- An IVM64 port of the PS/PDF interpreter *ghostscript* [27],
- A collection of applications that are of interest for the iVM project (*ivm-apps* [14]), such as programs to work with jpeg and tiff formats, compression libraries (zlib), booting routine and some others.

One of the challenges when testing real programs has been the lack of file systems for the IVM64 architecture, as the I/O is limited to some specific instructions without support for file streams. 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 to modify the sources codes. This is the case of the *boxing* benchmark for which authors has embedded the test images into the C code (*static_boxing* test). The *ghostscript* port uses the ROMFS support included in this package in order to insert a fixed PDF for testing.

For other cases, like the codes from the LLVM testsuite, an in-memory folderless file system generator [15] 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 sources, that can access the files without requiring modifications. This features relies on the linker's ability to supersede the libc file primitives that are replaced by those provided together the filesystem. This is done by the `ld` script that, in case of global symbol repetition, discards all occurrences except the first one.

Table 9 shows the compiler performance for the *static_unboxer* test [26] with various optimization levels. Codes have been compiled and executed with the target version 2.1 based on GCC 10.2.0, on a

Intel Core i7-3770 @3.40GHz server running a Linux kernel 4.15.0. Binaries were generated with the ivm implementation v0.37 [13]. The execution time spent by different external tools is shown in table 10 for the *static_unboxer* test compiled with the default optimization level.

Optimization level	Executed instructions (in millions)	Binary size (in KBytes)	space × time factor†
`-O3`	**14408**	1466	22
`-Os`	16036	**1279**	21
`-O2` (default)	14968	1369	**20**
`-O0`	45171	1700	77

† executed instructions (millions) × size (KB)/1e6, rounded to the nearest integer

Table 9: Compiler performance for the *static_unboxing* test.

Tool	Execution time
Assembler (`ivm as`)	10 s.
Emulator `vm.c`	85 s.
Yet another (fast) IVM emulator	34 s.

Table 10: Performance of the external tools for the *static_unboxing* test.

6 Conclusions

Designing a C compiler back-end for a pure stack based machine, such as the IVM64 architecture, poses some challenges since most mainstream C compilers are targetted for register based machines. The GCC port to the IVM64 architecture has resulted in a robust and efficient C cross-compiler, that meets all the specifications contemplated in the iVM project. The compiler has been tested and validated not only with project-specific applications but also with an ample set of benchmarks. The generated code shows an outstanding performance both in the number of executed instructions and the binary size. In addition to the back-end itself, the standard C library was also ported, and some other helper utilities were developed, such as fast emulators and an in-memory file system generator.

References

[1] Immortal Virtual Machine - solving the problem of file format and infrastructure obsolescence. Eurostars Cut–off 9 project (Reference Number: 12494), 2018-2021.

[2] Bjørn H Brudeli. A holistic approach to digital preservation. In *SMPTE 2014 Annual Technical Conference Exhibition*, pages 1–11, 2014.

[3] Richard M Stallman and the GCC Developer Community. GNU Compiler Collection internals (for gcc version 10.2.0). *Free Software Foundation*, 2020.

[4] Krister Walfridsson. *Writing a GCC back end*, 2017. `https://kristerw.blogspot.com/2017/08/writing-gcc-backend_4.html` (retrieved Dec 1, 2020).

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

[6] Hans-Peter Nilsson. Porting the GNU C compiler to the CRIS architecture. *`ftp://ftp.axis.se/pub/axis/tools/cris/misc/rapport.pdf`*, 1998.

[7] Jan Parthey. Porting the GCC-Backend to a VLIW-Architecture. 2004.

[8] Harry Gunnarsson and Thomas Lundqvist. Porting the GNU C compiler to the Thor microprocessor. Master's thesis, Saab Ericsson Space AB, Sweden, 1995.

[9] *ZPU - The worlds smallest 32 bit CPU with GCC toolchain*, 2009. https://opencores.org/projects/zpu (retrieved on Dec 1, 2020).

[10] Ivar Rummelhoff. Formal specification of VM and I/O devices and description validation (Eurostars Programme, iVM project). Technical report, Norwegian Computing Center, Oslo, Norway, 2021.

[11] Thor Kristoffersen. A guide to building the Immortal Virtual Machine. Technical report, Norwegian Computing Center, Oslo, Norway, 2020.

[12] Thor Kristoffersen. The Immortal Virtual Machine Instruction Set Architecture. Technical report, VirtuMa Solution 3.3 Part a, 2020.

[13] Ivar Rummelhoff. IVM implementation (v0.37), 2020. https://github.com/immortalvm/ivm-implementations.

[14] Liabø Ole and Bjarte M Østvold. iVM-app modules, 2020. https://github.com/immortalvm/ivm-apps.

[15] Eladio Gutierrez, Sergio Romero, and Oscar Plata. IVM filesystem generator, 2020. https://github.com/immortalvm/ivm-fs.

[16] Eladio Gutierrez, Sergio Romero, and Oscar Plata. Yet another (fast) IVM emulator, 2020. https://github.com/immortalvm/yet-another-ivm-emulator.

[17] Fabrice Bellard. QEMU: a fast and portable dynamic translator. In *USENIX annual technical conference*, volume 41, page 46, 2005.

[18] Sergio Romero, Eladio Gutierrez, and Oscar Plata. A QEMU port for the IVM64 architecture, 2021. https://github.com/immortalvm/qemu-ivm.

[19] Corinna Vinschen and Jeff Johnston. The Red Hat newlib C library. https://sourceware.org/newlib/ (retrieved on Mar 1, 2021).

[20] Jeremy Bennett. *Howto porting newlib: A simple guide*, 2010. https://www.embecosm.com/appnotes/ean9/ean9-howto-newlib-1.0.html (retrieved on Mar 1, 2021).

[21] D A Gwyn. Alloca – a (mostly) portable public-domain implementation, 1986.

[22] Richard M Stallman and the GCC Developer Community. Using the GNU Compiler Collection (GCC) (for gcc version 10.2.0). *Free Software Foundation*, 2020.

[23] Eladio Gutierrez, Sergio Romero, and Oscar Plata. The IVM64 compiler, 2021. https://github.com/immortalvm/ivm-compiler.

[24] Fabrice Bellard. Tiny C Compiler. http://bellard.org/tcc (retrieved on Mar 1, 2021).

[25] LLVM testsuite. https://github.com/llvm/llvm-test-suite.

[26] Liabø Ole. Boxinglib, 2020. https://github.com/immortalvm/boxing.

[27] Thor Kristoffersen. Ghostscript on IVM64. Technical report, VirtuMa Solution 3.3 Part b, 2020.