Implementation and comparison of register allocation to translate `LLVM--` to x86

William Welle Tange

2023

Contents

1 Introduction

Compilation refers to the process of translating from one language to another, most often from a high-level programming language intended for humans to work with, to machine- or bytecode intended to be executed on a target architecture. This process can be divided into several distinct phases, which are grouped into one of two stages colloquially referred to as the *frontend* and *backend*, the former translating a high-level programming language to an *intermediate representation* (IR) and the latter translating IR to executable machine code of a target architecture or bytecode of a target *virtual machine* (VM).

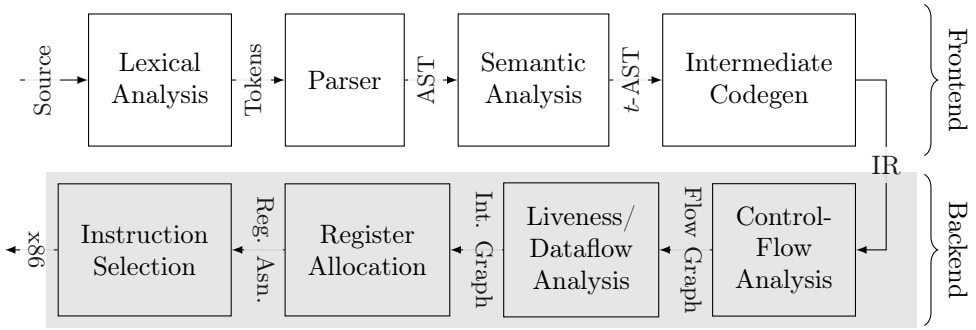

Figure 1: Compiler phases, backend highlighted

Most operations of a general-purpose programming language are translated to a set of control, logic, and arithmetic instructions to be executed sequentially on a computer processor: a single circuit/chip, referred to as the *central processing unit* (CPU), the design of which has varied and evolved over time.

Most CPUs are *register machines*, in that they use a limited set of *general-purpose registers* (GPRs) to store working values in combination with *random access memory* (RAM) for mid-term, and other I/O peripherals for long-term storage. This can largely be attributed to performance, as register machines routinely outperform *stack machines* (Shi et al., 2008) that are often used in VMs. Although register machines generally also have a stack available, as being limited to mere bytes of storage is simply unfeasible for most large scale applications, using it compared to GPRs is several orders of magnitude slower. Because of this, a crucial part of the backend stage for an optimizing compiler is assigning each variable of the source program to a GPR in such a way that maximizes performance without sacrificing correctness.

The process of assigning each variable to a GPR is referred to as *register allocation*, and can be approached in several different ways. This paper will seek to implement graph coloring and linear scan and evaluate them in terms of runtime performance after compilation. The primary sources will be *Modern Compiler Implementation in ML* (Appel, 1997) and *Compilers: Principles, techniques, and tools* (Aho et al., 2014), in addition to publications concerning the linear scan approach.

2 Control Flow Analysis

The backend of a compiler takes some form of IR as input, usually a linear sequence of instructions for each separate function. This representation is close to the level of an actual processor by design, but it isn't immediately useful for the further analysis steps needed to generate optimized code for the target architecture. The control flow of a given program refers to the order in which instructions are executed. While the flow of most instructions is linear, in the sense that the next instruction executed is located immediately after, some transfer the flow of execution elsewhere or terminate it.

A continuous flow of instructions is referred to as a *basic block*, defined as a sequence of instructions with no branches in or out except for the first instruction (referred to as a *leader*, immediately following either the function entry or label within it) and the last (referred to as a *terminator*, as it either terminates or transfers the flow of execution). In order to represent the order in which instructions are executed, a *control flow graph* (CFG) is introduced. It is a directed graph whose edges represent transfer of control. The type of node varies over the source material, with CFGs of the Appel text constructed over individual instructions (Appel, 1997) and the Aho text over basic blocks (Aho et al., 2014). Either way, leaders have a set of predecessors and terminators a set of successors associated with them.

Terminators have one or two successors in the case of branching or none at all in the case of function exit. Unconditional branching always transfers control to the block labelled, meaning only one successor follows, whereas conditional branching could transfer to either of the two, but because control flow analysis is not concerned with data, both are considered as possible successors.

Successors of node n are denoted $succ[n]$, and while each node also has predecessors associated, which consist of an unbounded number of nodes from which control may be transferred, it isn't particularly useful in the following analysis.

2.1 Building a graph

With an input stream of instructions

Each function defined in an LLVM program is constructed with the help of

3 Liveness Analysis

Translating IR with an unbounded number of variables to a CPU with a bounded number of registers involves the process of assigning each variable a register such that no value that may be needed in the future is overwritten. Variables that are in use at a given program point are considered *live*, and although variables can be assigned the same register, variables that are live at the same time (i.e. at intersecting program points) cannot, in which case they are also said to be in interference with one another. Variables that are not in interference can be assigned the same register, and finding the precise points at which any variable is live is trivial for linear sequences of instructions. However, when conditional branching is introduced, deriving the path of execution becomes undecidable.

For instance, suppose a function that calls another:

```
1 define i32 @countcall(i32 %x0) {
2   %x1 = add i32 %x0, 1
3   call ptr @subproc()
4   ret i32 %x1
5 }
```

Because of the halting problem, static analysis cannot determine if the call to `@subproc` will return for every possible implementation. So when assigning `%x1` a register it is undecidable whether the variable must be live in the last return instruction, i.e. needs to live across the call to another. Because of this, any sound approach to liveness analysis will be an approximation.

In some cases it is simply impossible to assign each variable its own register without conflict along interference edges, in which case one or more variables need to be assigned to memory instead. This is also referred to as *spilling* the variable/register to the stack, or the variable itself is referred to as *spilled*.

A sound albeit very naive approach is to consider every variable live at every program point, such that every variable is in interference with one another, producing a fully connected interference graph. This forces every variable to be assigned a different register, which is a viable approach for small programs with fewer variables than the set of assignable registers. Such a heuristic is greedy in the sense that it picks an assignment known to be safe for the least amount of preprocessing possible. However, this can be a very inefficient assignment at runtime. When the number of variables is greater than the number of assignable registers, the variable is spilled. A number of n variables greater than k working registers causes $n - k$ variables to be spilled to memory, which can greatly reduce performance, but allows for translation in linear time.

3.1 Dataflow Analysis

Another approach, which is a much more precise approximation, is a specific variant of the dataflow analysis as described in *Modern Compiler Implementation in ML* (Appel, 1997) and *Compilers: Principles, Techniques, and Tools* (Aho et al., 2014). In general, dataflow analysis is the process of finding the possible paths in which data may propagate through various branches of execution. While several applications of this exist (like constant propagation, reaching defnitions, available expressions etc.), one that is immediately beneficial in the case of liveness analysis is one that traverses a CFG in the reverse order of execution (i.e. *backwards* flow), and extracts any variable that *may* be used in execution (also referred to as *backwards may* analysis).

This algorithm calculates which program points each variable may be accessed from with some conservative constraints known to maintain correctness. Specifically, these are the transfer and control-flow constraints. The transfer constraint is based on a *transfer function* that describes how liveness is affected across instructions. For each instruction, there is a transfer function that describes how liveness changes from one point to the one immediately after. For example, as an arithmetic operation needs to be assigned a new temporary variable, the liveness of a new variable is propagated to all instructions executed subsequently. This is done by applying the transfer function to the current *live-out* set to stop further propagation of variables defined by the currently visited instruction.

$$in\,[n] = use\,[n] \cup (out\,[n] - def\,[n]) \tag{1}$$

with the $use[n]$ set being defined as any variables that may be used and the $def[n]$ as the set of variables defined in node n. Because the chosen IR is in SSA form the $def[n]$ either consists of one variable or equal to $\emptyset$. The $use[n]$ set is effectively unbounded as some instructions take any m variables as parameters.

Control-flow constraints on the other hand propagate the use of variables to previously executed instructions, expecting these to be defined somewhere further up the CFG. This is done by propagating the union of the *live-in* set associated with all immediate successor nodes. This is also referred to as the *meet operator*, whose operator depends on the type of dataflow analysis as well, but for liveness analysis a union is performed on the *live-in* sets of any successive nodes:

$$out\,[n] = \bigcup_{s \in succ[n]} in\,[s] \tag{2}$$

Initially, two sets are associated with each instruction: the *live-in* and *live-out* sets, which are the sets of variables that are live respectively before and after execution. Then the two equations above are applied iteratively until a fixed point is reached, i.e. an invariant point where neither $in[n]$ or $out[n]$ is changed for all instructions n.

The simplest equations to implement are the `def` and `use` functions, as all of the values of interest are located immediately within the instruction itself and not hidden behind some layer of indirection:

```
let def (s : S.SS.t) (insn : Cfg.insn) =
  match insn with Insn (Some dop, _) -> S.SS.add dop s | _ -> s

let use (s : S.SS.t) (insn : Cfg.insn) =
  let op o s = match o with Ll.Id i -> S.SS.add i s | _ -> s in
  let po s o = op o s in
```

```
  match insn with
  | Insn (_, AllocaN (_, (_, o)))
  | Insn (_, Bitcast (_, o, _))
  | Insn (_, Load (_, o))
  | Insn (_, Ptrtoint (_, o, _))
  | Insn (_, Sext (_, o, _))
  | Insn (_, Trunc (_, o, _))
  | Insn (_, Zext (_, o, _)) ->
      op o s
  | Insn (_, Binop (_, _, l, r))
  | Insn (_, Icmp (_, _, l, r))
  | Insn (_, Store (_, l, r)) ->
      op l s |> op r
  | Insn (_, Call (_, _, args)) -> List.map snd args |> List.fold_left po s
  | Insn (_, Gep (_, bop, ops)) -> List.fold_left po (op bop s) ops
  | Insn (_, Select (c, (_, l), (_, r))) -> op c s |> op l |> op r
  | Insn (_, PhiNode (_, ops)) -> List.map fst ops |> List.fold_left po s
  | Term (Ret (_, Some o) | Cbr (o, _, _)) -> op o s
  | _ -> s
```

Where `S.SS` is a `Set.S` module built over the `symbol` type found in `lib/symbol.ml`:

```
type symbol = string * int
(* ... *)
module SS = Set.Make (struct
  type t = symbol
  let compare (_, n1) (_, n2) = compare n1 n2
end)
type set = SS.t
```

With the actual fixed-point iteration performed as follows:

```
let dataflow (insns : Cfg.insn list) (ids : Cfg.G.V.t array) (g : Cfg.G.t) =
  let insns = List.mapi (fun i v -> (i, v)) insns |> List.rev in
  let in_ = Array.init (List.length insns) (fun _ -> S.SS.empty) in
  let out = Array.init (List.length insns) (fun _ -> S.SS.empty) in
  let rec dataflow () =
    let flowout = (* ... *)
    let flowin = (* ... *)
    let flow changed insn = changed || flowout insn || flowin insn in
    if List.fold_left flow false insns then dataflow () else (in_, out)
  in
  dataflow ()
```

The `flowin` function correponds to the *live-in* equation (1) and is implemented as follows:

```
let flowin (i, insn) =
  let newin = S.SS.union (use insn) (S.SS.diff out.(i) (def insn)) in
  let changed = not (S.SS.equal newin in_.(i)) in
  if changed then in_.(i) <- newin;
  changed
```

And the `flowout` function which correponds to the *live-out* equation (2) implemented as follows:

```
let flowout (i, _) =
  let newout =
    let succ = Cfg.G.succ g ids.(i) in
    List.fold_left
      (fun s v -> S.SS.union s in_.(Cfg.G.V.label v))
      S.SS.empty succ
  in
  let changed = not (S.SS.equal newout out.(i)) in
  if changed then out.(i) <- newout;
  changed
```

3.2 Interference Graph

The purpose for conducting dataflow analysis as above is finding variables that may be assigned the same register. This is done by building an interference graph, which is an undirected graph, whose nodes represent variables and edges that signify interference between them, i.e. variables a and b live at overlapping program points is represented with an edge (a, b).

Constructing an interference graph only depends on the *live-out* set and type of instruction. If the instruction defines a variable, said variable is in interference with all variables in the *live-out* set. There is one exception however: according to the Appel text, move instructions (i.e. phi nodes in the case of SSA form) are given special consideration. The purpose of phi nodes is to copy/move a certain value from a certain predecessor, so they are not necessarily in conflict for being live at the same time. Rather it would often benefit if they were assigned the same register to spare unnecessary moves.

Because of this, for any phi node of the form

$$a = \Phi(b_1, ..., b_n)$$

add edges to all *live-out* variables not in B:

$$\forall b_j \in (out[i] \setminus B), add\_edge(a, b_j) \text{ where } B = \{b_1, ..., b_n\}$$

For any other instruction that defines a variable a

$$\forall b_j \in B, add\_edge(a, b_j).$$

Although the Appel text (Appel, 1997) describes interference of variables with concrete registers as well as overlapping variables, this isn't considered in this implementation.

4 Graph Coloring

Once an interference graph is constructed, the actual assignments can be found using graph coloring. Although this has long been known to be NP-complete, the heuristic as introduced in both the Appel and Aho et al. texts is a linear time approximation to this problem. It is based on an iterative approach wherein nodes known to be colorable are removed until either an empty graph remains, in which case the original graph G is k-colorable or nodes with more than k neighbours remain. In this case a node is chosen to be spilled to the stack and removed. This is then repeated.

Let k be the number of working registers available on the target architecture. After building the interference graph G, a node n with fewer than k neighbors is chosen. As n has at most $k - 1$ neighbors, it can be removed safely, effectively simplifying G. It is pushed to a stack to preserve the order in which they are removed so that G can be rebuilt and the corresponding registers can be assigned correctly once a k-colorable assignment is found.

4.1 Coloring by simplification

4.2 Coalescing

Coalescing is the process of eliminating moves/copies of data from one GPR to another by combining their interference graph nodes. This is similar to but not the same as the lack of interference edges between variables that are subject to move operations. This is because variables a and b may still be assigned different registers or even spilled if, for instance, either of them are of significant degree. Coalescing joins nodes a and b to node ab preserving the edges of both to maintain soundness.

Since all edges are preserved, the resulting node ab may be of a much higher degree. Because of this, only strategies that produce a k-colorable graph are worth considering, as additional spills negate the purpose entirely.

5 Linear Scan

6 Instruction Selection

6.1 `LLVM--` instruction set

The intermediate representation emitted by the frontend of a compiler serves as a stepping stone independent of the target architecture. The `LLVM` infrastructure is the industry standard in terms of bridging this gap and was consequently the library used to translate the semantically annotated abstract syntax tree to executable machine code in the 2022 compilers course. As this project is an expansion on this, it follows naturally to build on this.

The instruction set used in this paper will be a union of the sets used in the 2022 and 2023 compilers courses in order to work as a drop-in replacement of LLVM for either of the two respective source languages: Tiger and Dolphin. This instruction set is a subset of the one used in practice, as, for instance, neither of the languages implemented support exception handling, floating point operations and so on, and instead only strive to cover the basics of compilers.

The instructions included are `trunc` has only been added in order to help cover more generated LLVM. The branching for most of these is trivial: after successful execution the flow of all but `br`, `ret` and `unreachable` will unconditionally attempt to execute the next instruction in memory (i.e. increment the instruction pointer by instruction length).

Because of this, these instructions are referred to as *terminators*, as their purpose is to disrupt what had otherwise been a linear flow from the beginning of this continus sequence of instructions, henceforth referred to as a *basic block*.

6.2 Translating `LLVM--` to x86

Translating each IR instruction to x86 correctly is a matter of eliminating unintended side-effects. Each LLVM– instruction is defined to have only one purpose, as concepts such as calling conventions, stack frames, or a `FLAGS` register are completely abstracted over in order to remain platform independent.

In contrast, the x86 *instruction set architecture* (ISA) is targeting a *complex instruction set computer* (CISC) family of processors, the instructions of which perform a much broader set of operations Appel, 1997, p. 190. This is is in part due to pipelining, i.e. an abstraction over the concrete implementation of the actual processor, which in turn is made for the sake of performance.

An example of this would be the division/remainder operation: since integer division is a non-trivial iterative process wherein both the quotient and remainder is needed throughout, the result of both of these is stored in the `%rax` and `%rdx` registers respectively. This means two operations are performed simultaneously regardless of which value is used, hence they need to be restored before executing the next instruction, as any variables assigned to `%rax` or `%rdx` will be overwritten.

add	addx	{}
mul	imul	{ %rax }
sdiv	idivx	{%rax, %rdx}
srem	idivx	{%rax, %rdx
call	callx	{%rax, *caller saved registers*}

6.3 Assessing Correctness

It is difficult to assert the correctness of a compiler. While writing simple unit tests is easy, guaranteeing the correctness of code generated for any given input is on a different order of magnitude in terms of complexity.

Validating the correctness of each instruction translation is a complicated task given how low-level it is. One approach is attaching a debugger and examining the state before and after the translated sequence of x86 instructions is executed. The result of applying a binary operation, for instance, should only affect the assigned register.

6.3.1 Debugger Harness

A more elaborate way of testing correctness of translation is to assert that there are no effects outside of those intended. One way of achieving this is to check if a group of x86 instructions representing a single LLVM– operation change any values except those expected. An approach to this is to attach a debugger, insert a breakpoint before every LLVM– instruction, and ensure that only the intended registers are altered.

To achieve this, only a few changes need to be made to the codebase. A debug flag is introduced, which inserts a label before every instruction by adding a `Breakpoint` variant of the `ins` type before the assembly emitted of that specific operation. In this a unique identifier as well as a list changed expected to be made is encoded. This includes a bitmask of GPRs that can possibly be changed (aside from the scrath registers of couse) as well as which parts of the stack may be modified. In addition to this, each of the labels are output with a `.globl` directive in the file header. A file is built with these extra symbols using the `-d` flag.

To validate that this works as intended,

7 Evaluation

While there are many approaches to assessing the quality of translation, one of the primary means is to simply measure the time taken to execute the code generated. Assuming the translation is sound given the previous steps taken to validate correctness in translating `LLVM--` to x86, it is a solid foundation for future optimizations. While there are certainly other factors worth measuring, only runtime performance is considered for this project.

7.1 Benchmarking

Since the target architecture is x86-64, all benchmarks are executed natively on the fastest processor available to be used consistently at the time of writing. This is to maximize the sample size and in turn reduce uncertainty, but the clock speed at which the benchmarks are performed is irrelevant to how fast the generated code is. Because of this, the metric recorded is the amount of CPU cycles spent executing from start to finish instead of actual time in seconds, as the time measure only works as a more imprecise estimate of the amount of underlying work/execution steps performed on the processor.

CPU cycles isn't a perfect measurement either, but it works to eliminate the clock speed as well as negate much of the time the scheduler allocates for other processes. Scheduling still has a negative impact on execution because of the cache misses caused by context switching, but outside of executing each program in immediate mode (which isn't possible on Linux or macOS) this is a sensible estimate. To further reduce context switching, benchmarks are made on a fresh restart with minimal processes running (Xorg etc.).

Additionally, all benchmarks are deterministic, meaning relevant values at all points across all runs are consistent. So while not completely equivalent due to address randomization, such noise should never leak to value space, and assuming values allocated are initialized before use, no remnants from other processes should leak to value space either. This means that the cycles measured over infinitely many runs will converge

towards a 'perfect' run with no cache misses as caused by context switching. Although realistically this will never happen in a lifetime, across n runs the run least affected by context switching will be the one with the least CPU cycles measured, so is the one most representative of the actual performance without noise.

Measurements are made with the `perf` utility, which relies on the *hardware counters* (HC) registers of modern microprocessors, which are special purpose registers meant to record performance data during execution. Because the analysis is integrated into the chip on which it is executing, very little overhead is incurred, and is therefore the go-to for estimating native performance on Linux. The `perf` subcommand is used, as it starts a process and attaches from the very beginning, in addition to padding a `-e` flag specifying that only cycles are meant to be recorded and the `-x` flag to help parse the output. Both min, avg and max are tracked and reported but only min are represented in the matrices below.

Each `.ll` file present in the `benches` directory is listed here in a table denoting the input paramater on the left column and type of allocator used.

7.1.1 `benches/fib.ll`

The 'dumb' fib is a very naive approach to finding the nth number of the Fibonacci sequence by calling itself recursively twice (decrementing n before each), which causes an exponential growth in function calls.

n	clang	simple 12	simple 2	briggs 12	briggs 2	linear	greedy 12	greedy 0
20	1.00	1.23	1.23	1.23	1.23	1.22	1.22	2.23
21	1.00	1.27	1.27	1.27	1.29	1.27	1.27	2.44
22	1.00	1.30	1.29	1.30	1.29	1.30	1.29	2.63
23	1.00	1.32	1.32	1.32	1.32	1.33	1.32	2.75
24	1.00	1.34	1.34	1.34	1.33	1.34	1.40	2.83
25	1.00	1.36	1.35	1.35	1.35	1.35	1.35	2.89
26	1.00	1.36	1.35	1.35	1.35	1.36	1.40	2.92
27	1.00	1.36	1.35	1.36	1.36	1.36	1.36	2.96
28	1.00	1.37	1.36	1.38	1.37	1.37	1.36	2.97
29	1.00	1.37	1.37	1.37	1.36	1.37	1.37	2.98
30	1.00	1.35	1.34	1.35	1.34	1.33	1.34	2.91
31	1.00	1.36	1.37	1.37	1.36	1.37	1.36	2.98
32	1.00	1.37	1.37	1.38	1.37	1.37	1.38	2.99
33	1.00	1.37	1.37	1.38	1.37	1.37	1.37	2.99
34	1.00	1.37	1.37	1.37	1.37	1.38	1.37	2.99
35	1.00	1.37	1.37	1.37	1.37	1.37	1.37	2.99
36	1.00	1.37	1.37	1.37	1.37	1.37	1.37	2.99

The above measures are significantly equivalent to one another, with all allocators (except greedy 0) seeming to converge towards being 37% slower than Clang. This isn't particularly surprising, as the x86 translation is more concerned with preserving callee saved registers than necessary in pushing each and every one despite not being in use. Also, instead of pushing with the `push` instruction it would be faster to simply subtract from the `%rsp` register directly and using `mov` directly.

What's more interesting and related to the allocation is the fact that greedy 0 is more than twice as slow. This is presumably because the `fib` function that has 7 variables can assign each of them to a register for the first six allocators listed. However, the last one (greedy 0), has no assignable registers, so is forced to start spilling from the very beginning. To illusidate this hypothesis, here is the same benchmark but only for the greedy of decreasing k assignable registers:

n	clang	greedy 12	greedy 8	greedy 6	greedy 4	greedy 2	greedy 1	greedy 0
20	1.00	1.25	1.23	1.27	1.36	1.41	2.20	2.24
21	1.00	1.27	1.27	1.32	1.42	1.46	2.42	2.46
22	1.00	1.29	1.30	1.35	1.46	1.51	2.57	2.62
23	1.00	1.32	1.32	1.38	1.50	1.55	2.69	2.74
24	1.00	1.35	1.34	1.46	1.54	1.58	2.78	2.83
25	1.00	1.35	1.35	1.41	1.54	1.59	2.83	2.89
26	1.00	1.36	1.36	1.42	1.55	1.60	2.87	2.93
27	1.00	1.36	1.36	1.43	1.56	1.61	2.89	2.95
28	1.00	1.36	1.37	1.43	1.57	1.62	2.92	2.97
29	1.00	1.36	1.37	1.44	1.57	1.62	2.92	2.98
30	1.00	1.36	1.37	1.43	1.57	1.63	2.93	2.98
31	1.00	1.37	1.36	1.43	1.57	1.62	2.93	2.99
32	1.00	1.37	1.36	1.43	1.57	1.63	2.92	2.98
33	1.00	1.37	1.36	1.43	1.57	1.63	2.94	2.99
34	1.00	1.37	1.37	1.43	1.57	1.63	2.94	2.99
35	1.00	1.37	1.37	1.43	1.57	1.63	2.93	2.99
36	1.00	1.37	1.36	1.43	1.57	1.63	2.94	2.99

7.1.2 `benches/ackermann.ll`

Another benchmark that derives highly from the recursive structure of `fib.ll` above is the Ackermann function.

7.1.3 `benches/factori32.ll`

Prime factorization of `i32`.

fib	clang	greedy	simple	linear
40	1.000000	1.438693	1.441270	0.73
41	1.000000	1.441035	1.440626	0.73
42	1.000000	1.447140	1.446094	0

7.2 Comparison to other work and ideas for future work

8 Conclusion

References

Aho, A. V., lan, M. S., Sethi, R., & Ullmann, J. D. (2014). *Compilers : Principles, techniques, and tools* (2nd ed., internat. ed.). Pearson Education Limited.

Appel, A. W. (1997). *Modern compiler implementation in ml*. Cambridge University Press.

Shi, Y., Casey, K., Ertl, M., & Gregg, D. (2008). Virtual machine showdown: Stack versus registers. *ACM transactions on architecture and code optimization*, *4*(4), 1–36.

A LLVM– instruction set

1. binary operations `add`, `and`, `ashr`, `lshr`, `mul`, `or`, `sdiv`, `srem`, `shl`, `sub` and `xor`
2. integer comparison `icmp` (conditions being `eq`, `ne`, `sge`, `sgt`, `sle` and `slt`)
3. memory/address operations `alloca`, `gep`, `load` and `store`
4. `mov` operations `gep`, `phi`, `ptrtoint`, `trunc`, and `zext`
5. control flow operations `br`, `call` and `ret`
6. a nullary block terminator `unreachable`