WASM Stack Machine

titzer@google.com
Last Updated: 2016-08-17

Motivation and Background

WASM is a new, portable, size- and load-time-efficient executable format suitable for compilation to the web. A serialized AST representation of functions allows simple and efficient verification and compilation for consumers of the binary code. Postorder serialization allows fast verification, fast decoding to the AST form, fast compilation, and in-place interpretation.

To decode WASM functions, a *decoding stack* is normally used. Different decoding tasks store different information in the decoding stack. For example, decoding WASM to an explicit AST uses a stack of AST nodes, while typechecking uses a stack of types, compiling uses a stack of register/stack locations, constructing compiler IR uses IR nodes, and an interpreter uses a stack of runtime values. See **Figure 1** for an example.

Main problem. WASM expressions that return `void` (or no value) still occupy decoder stack space. For decoders which build an AST or compiler IR push and pop fragments of their respective representations, this is not a problem. However, the extra decoder stack space is significant for verifiers, interpreters and baseline compilers, since they typically cannot tell whether a given value will be consumed until reaching the end of a block. Very large basic blocks can introduce a lot of overhead.

Proposal. This document proposes WebAssembly code target a stack machine with structured control flow constructs.

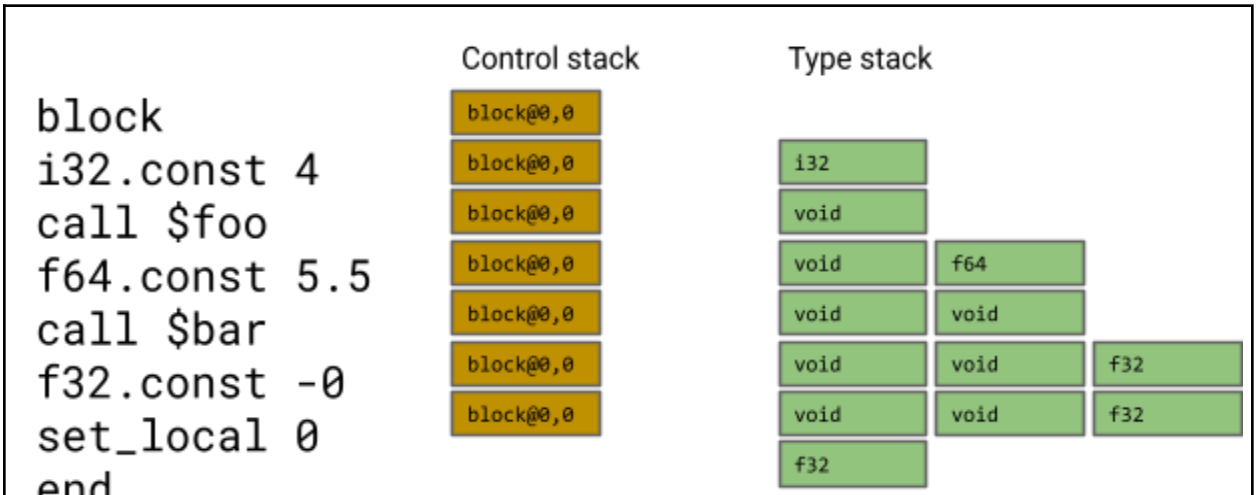

Figure 1. Example decoding stack for verification at each instruction. Each line shows the control and type stack state during verification as the verifier advances (down) through the

bytecodes.

A step forward with drop, set_local, and store_memory

To reduce the need to store the equivalent of void values or unused values produced by operations evaluated for side-effect only, https://github.com/WebAssembly/design/pull/711 introduced the concept of `drop` and `tee_local`, while also changing the semantics of `set_local` to no longer return the value stored to a local.

This PR, part of binary version 0xC:

- **Introduced** `drop`, a new operator that discards the value produced by evaluating an expression
- **Changed** `set_local`, so that it now consumes the value assigned to a local variable instead of returning it
- **Introduced** `tee_local`, a new operator that inherits the prior semantics of set_local, assigning a local and also returning the value
- **Required** `void` expressions to appear only at the beginning of blocks

The additional restriction of requiring `void` expressions to appear only at the *beginning* of blocks or loops means that blocks always consist of some number of `void` expressions followed by at most one non-void expression. Blocks and expressions still form ASTs; they simply come with a type restriction for their first expressions. With that restriction, it is possible to save decoder stack space by not pushing anything onto the decoder stack for `void` expressions. That translates directly into better register allocation for baseline compilation, smaller value stacks for interpreters, and smaller type stacks for verifiers.

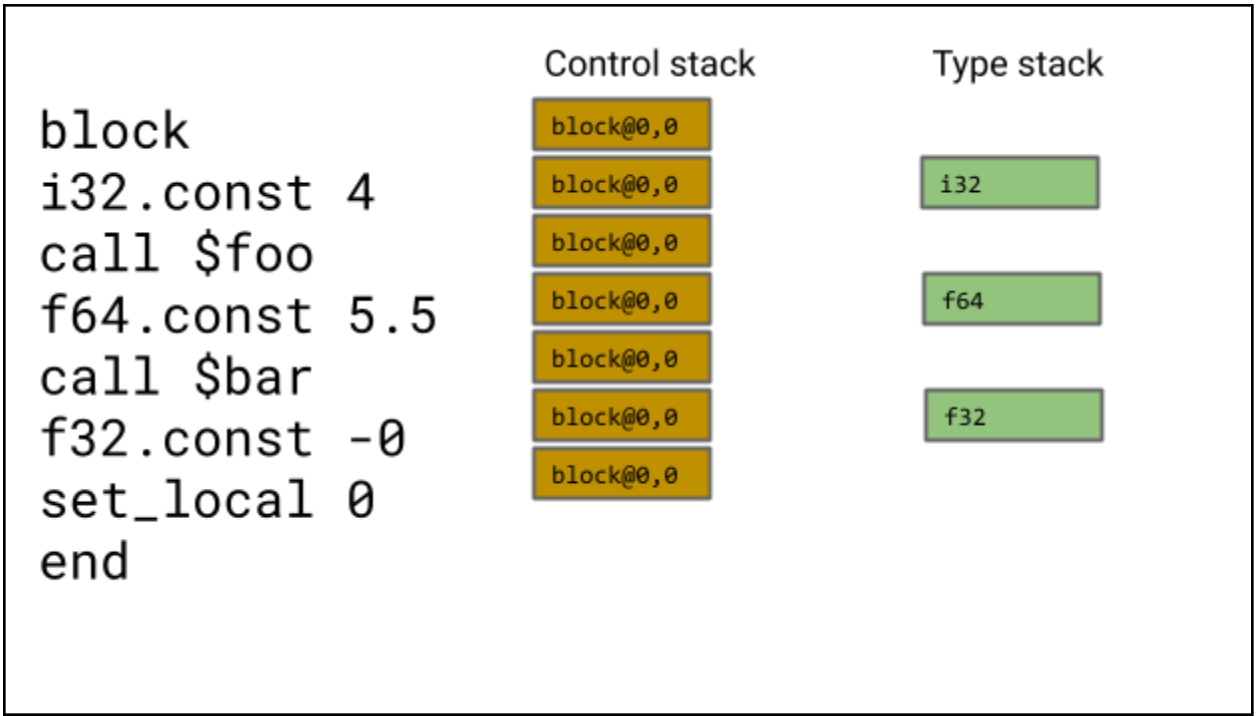

Figure 2. Example decoding stack for verification at each instruction with the stack machine (compare to Figure 1). Each line shows the control and type stack state during verification as the verifier advances (down) through the bytecodes.

Void in the middle: a stack machine

What about allowing `void` expressions to appear anywhere, even between the operands to a binary operator? That (small) generalization essentially turns WASM into a stack machine. A stack machine has a natural semantics for multi-value `block`, `if`, `call`, and `return`. When we eliminate implicit popping off the stack from blocks, then it also has a straightforward execution semantics.

Understanding the WASM Stack Machine

The WASM stack machine is a simple generalization of the postorder encoding. The machine has a *program counter* and a *value stack*. Individual operators such as `i32.add`, `f32.mul`, etc., push and pop values from the value stack, while control constructs change the program counter. Conceptually, executing a program consists of executing the operator at the program counter repeatedly until the program halts, with no reference to an AST.

Despite *seeming* entirely different than a postorder encoding of ASTs, the stack machine actually is only a simple generalization. The stack machine allows *more programs* than the previous AST formulation, while the execution of existing AST programs maps onto the stack machine essentially *unchanged*. That means existing AST producers can continue to work, but means that consumers need to deal with a broader class of potential programs.

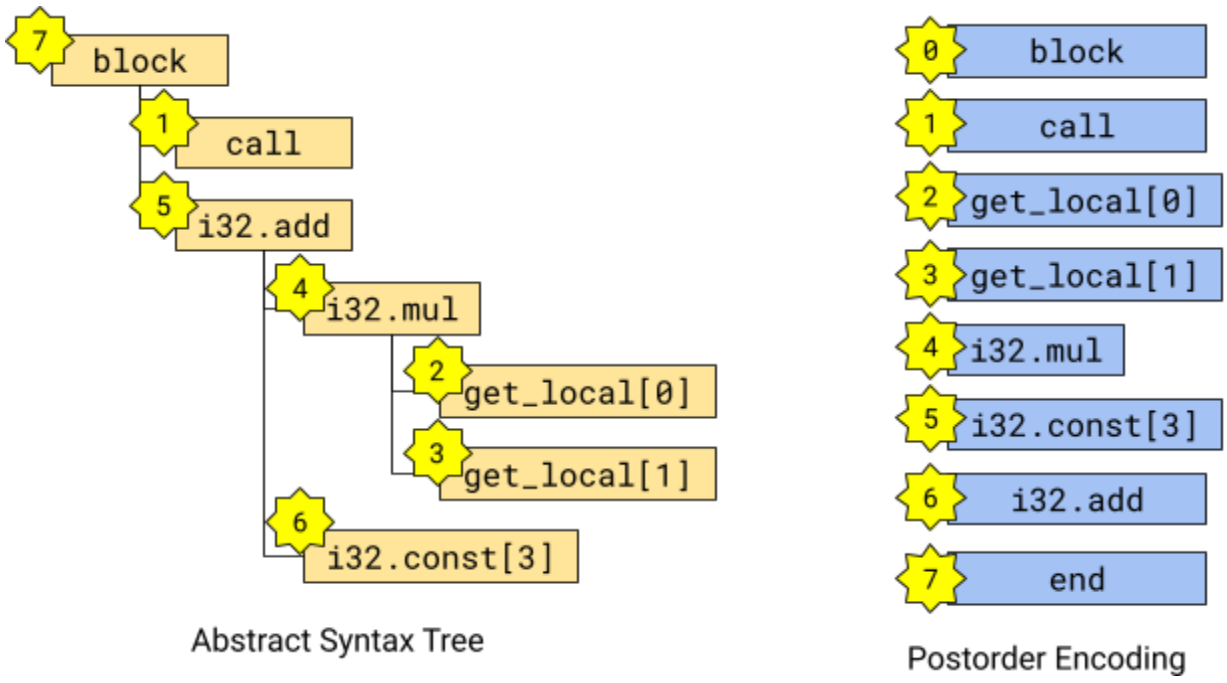

Figure 2. Abstract syntax tree and its postorder encoding with a block. Notice how the execution order is explicitly reflected in the postorder encoding.

Figure 2 illustrates how the postorder encoding of an AST reflects the execution order. The same property is preserved in the stack machine. In fact, the code in Figure 2 is valid stack machine code as well. **Figure 3** shows how the stack machine generalizes postorder ASTs. Imagine trying to construct a more complex program from the one given in Figure 3 by simply inserting the green void expression at various places in the original program. As we can see, the use of auxiliary blocks allows constructing ASTs for most locations, but there is no valid AST (without new constructs or looser rules) for locations between an operator and its last operand.

In contrast, the stack machine allows inserting this void expression code *anywhere* without auxiliary blocks.

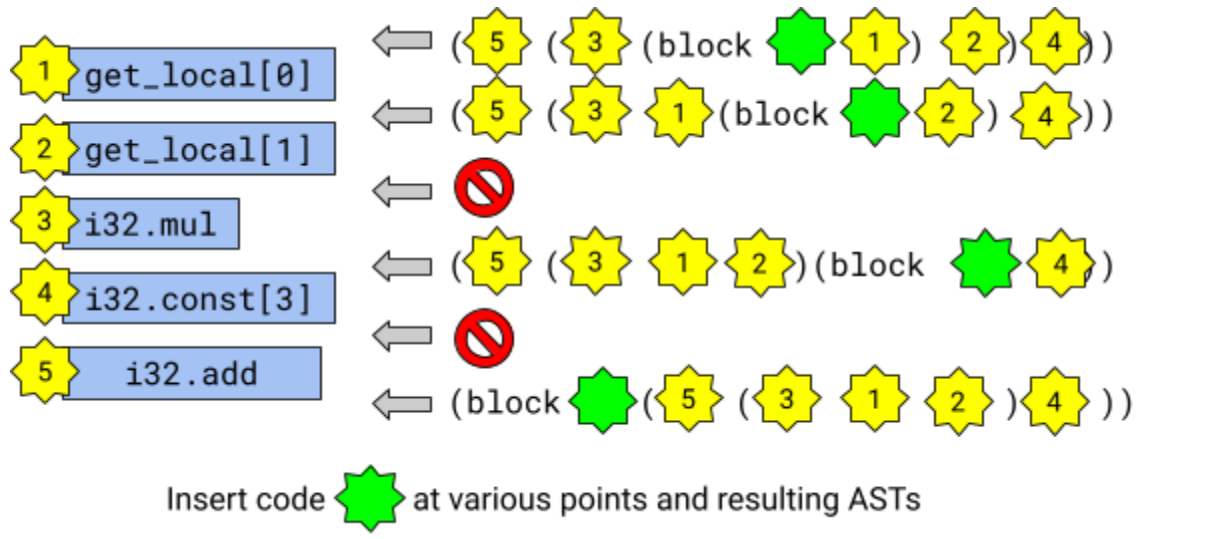

Figure 3. Insertion points for side-effecting operations demonstrate the increased generality of the stack machine versus using blocks for side-effects.

Like ASTs, the stack machine offers only structured control flow operations: `block`, `loop`, `if`, `br`, `br_if`, and `br_table`. The encoding of these constructs is unchanged in the stack machine. The execution semantics are actually simplified to operate only on the value stack and a *virtual control stack*. The main difference is that only the control constructs that alter the value stack are the explicit branches `br`, `br_if`, and `br_table`. The `block`, `if`, `else`, and `end` constructs do not implicitly pop any values.

Verifying Stack Machine Code

WebAssembly has always held efficiency of verification as a primary goal. Verifying post-order encoded ASTs is done with a decoding stack containing types, as mentioned before. It is not necessary to construct an explicit AST data structure to perform verification. With the stack machine, there is a straightforward extension of the decoding-stack based algorithm. The main new changes to verification are:

- All branches to the end of a `block` must have the same arity and same types, including the implicit fall-through to `end`.
- The true block of `if-end` constructs must leave the stack at the same height as when the if was entered.
- The true and false blocks of `if-else-end` constructs must leave the stack at the same height with the same types.

Verifying multi-values: The new verification rules handle blocks and if-else-end constructs that leave multiple values on the stack. Calls and returns can now naturally take multi-values as well; a call to a function that returns multiple values simply leaves those return values on the stack after the call returns.

Executing Stack Machine Code

Execution of the stack machine is pretty straightforward. As with an in-place interpreter for postorder WASM code (such as that built for the V8 interpreter), a program counter maintains the execution position within the code. A virtual control stack keeps track of `blocks` and `if` constructs as they are entered and exited. The control stack is used to determine stack adjustments for explicit branches. Execution of "simple" operators that simply pop their arguments off the stack and push their results is just as the postorder interpreter. The execution of control operators is simply:

- **`block`** or **`loop`**: push a new entry onto the control stack, recording the current value stack height.
- **`end`**: pop the control stack.
- **`if`**: push a new block onto the control stack, pop the condition off the value stack, and if the condition is not 0, fall through. Otherwise goto the matching `else` or `end` for this `if`.
- **`else`**: go to the matching `end` for the `if` on the top of the control stack.
- **`br`**: pop the result value(s) off the value stack, then pop up to the height of the target block, then push the result value(s), then goto the end of the corresponding block.
- **`br_if`**: pop the condition off the stack, then conditionally perform a `br`.
- **`br_table`**: pop the key off the stack, then perform a multi-way branch.

Proposal Status

- The prerequisite operators `drop` and `tee_local` have already been added to the binary_0xc branch on GitHub: https://github.com/WebAssembly/design/tree/binary_0xc
- The stack machine has been prototyped in the V8 WASM implementation, both the interpreter and the compiler, along with multi-value support: https://codereview.chromium.org/2176653002/
- It's fully implemented in a branch of the spec interpreter: https://github.com/WebAssembly/spec/tree/stack/ml-proto
- The binary encoding branch for 0xC is in the process of being updated for orthogonal changes to `block`, `branch`, `call`, and `return` arities.
- Implementation partners from Google, Mozilla, Microsoft and Apple have discussed and tentatively agreed on the stack machine.