a86 Interpreter Semantics

Pierce Darragh

June 3, 2022

This document describes the semantics used in the implementation of the a86 interpreter.

1 Syntax

We provide a definition of the abstract syntax of the a86 language.

									inst	::=	`Label`	**loc**		
r64	::=	`rax`	|	`rbx`	|	`rcx`	|	`rdx`		|	`Ret`			
	|	`rbp`	|	`rsp`	|	`rsi`	|	`rdi`		|	`Call`	(**loc** | **reg**)	
	|	`r8`	|	`r9`	|	`r10`	|	`r11`		|	`Mov`	(**reg** | **off**)	(**reg** | **off** | **int**)
	|	`r12`	|	`r13`	|	`r14`	|	`r15`		|	`Add`	**reg**	(**reg** | **off** | **int**)
										|	`Sub`	**reg**	(**reg** | **off** | **int**)	
r	::=	**r64**	|	`eax`						|	`Cmp`	(**reg** | **off**)	(**reg** | **off** | **int**)	
										|	`Jmp`	(**loc** | **reg**)		
f	::=	`CF`	|	`ZF`	|	`SF`	|	`OF`		|	`Je`	(**loc** | **reg**)		
										|	`Jne`	(**loc** | **reg**)		
l	::=	*labels*								|	`Jl`	(**loc** | **reg**)		
										|	`Jg`	(**loc** | **reg**)		
z	::=	*integers*								|	`And`	(**reg** | **off**)	(**reg** | **off** | **int**)	
										|	`Or`	(**reg** | **off**)	(**reg** | **off** | **int**)	
bc	::=	*integers 0 through 63 inclusive*								|	`Xor`	(**reg** | **off**)	(**reg** | **off** | **int**)	
										|	`Sal`	**reg**	**bc**	
off	::=	`Offset` **r** **z**								|	`Sar`	**reg**	**bc**	
										|	`Push`	(**int** | **reg**)		
prog	::=	`Program` (**inst** ...)								|	`Pop`	**reg**		
										|	`Lea`	(**reg** | **off**)	**loc**	

2 Semantics (Informal)

Before formalizing our semantics, we provide an informal specification. These definitions are based on existing documentation and code. A formal semantics is presented in the next section.

2.1 Program

First, we define a *program*. A program is a list of instructions, where:

- The list is not empty.
- The first instruction is a `Label`, which will be used as the entry point of the program (i.e., it is where execution begins).
- Two `Label` instructions in the same program cannot use the same label name.

2.2 Registers

There are sixteen 64-bit registers, each corresponding to one of the names in **r64**. The special register reference `eax` refers to the lower 32 bits of the `rax` register.

There is, technically, an additional register, `rip`, known as the "instruction pointer". This register keeps track of the memory address to go to for the next instruction. We do not expose this register directly, so no more is said of it here.

2.3 Flags

There are four single-bit registers, called *flags*, each corresponding to one of the names in `f`. The flags are used for arithmetic and comparison operations:

- `OF` — Overflow Flag

  Set when...

  ...*adding* two numbers with the same sign bit and the result has a different sign bit.

  ...*subtracting* a negative number from a positive number and the result is negative.

  ...*subtracting* a positive number from a negative number and the result is positive.

- `SF` — Sign Flag

  Set to the value of the sign bit of the result.

- `ZF` — Zero Flag

  Set if the computed result is exactly `0`.

- `CF` — Carry Flag

  Set if the (unsigned) arithmetic operation required an extra bit.

We say a flag "is set" if the value `1` is stored in it, or the flag "is clear" or "is unset" if the value `0` is stored in it instead. We may also use these terms as verbs, i.e., "to set" a flag means to store `1` in it and so on. Note, however, that "set" is overloaded as a verb, since it can also be used to indicate storing a specific value (e.g., "the flag is set to the value of <some computation>").

2.4 Memory

All a86 programs also run with some limited amount of register-external memory called the *stack*. The stack starts at the highest available address space and "grows downwards", which means that adding something to the stack *decrements* the pointer to the current position in the stack.

2.5 Instructions

There are 20 supported instructions in a86, which work as follows:

- `Label l` creates a new label named `l` that points to the next instruction.
- `Ret` pops an address from the stack and jumps to it.
- `Call dst` pushes the next instruction address onto the stack, then jumps to the address indicated by the label or register in `dst`.
- `Mov dst src` moves the contents of/value at `src` into `dst`.

  NOTE: Either `dst` or `src` may be an offset, but not both.

- `Add dst src` adds `src` to `dst` and writes the result to `dst`.

  Imagine we have only 4-bit numbers (instead of the 64 bits the a86 actually uses). Suppose we add `0111` and `1010`:

  ```
    0111
    1010   +
  -------
   10001
  ```

  Due to carries, we have ended up with a 5-bit result, but we can only have 4 bits to a number. The most-significant bit will be lost, but the `CF` flag will be set to indicate a carry has occurred.

  Here are some example additions of other 4-bit numbers:

OF	SF	ZF	CF	Left Operand	Right Operand	Result
0	0	0	1	0111	1010	0001
0	0	0	0	0010	0001	0011
0	1	0	0	1100	0001	1101
1	1	0	0	0111	0001	1000
0	1	0	1	1100	1101	1001

NOTE: The adding circuits do not even think about whether the numbers are signed or unsigned, because the addition is performed the same in both cases.

- `Sub dst src` subtracts `src` from `dst` and writes the result to `dst`.
- `Cmp a1 a2` compares `a1` to `a2` by subtracting the former from the latter and sets the flags according to the result:

  `CF` is set if an extra bit was needed to complete the computation.

  `ZF` is set if `a2 - a1` $= 0$.

  `SF` is set if `a2 - a1` < 0.

  `OF` is set if either...

  - ...`a1` is negative, `a2` is positive, and `a2 - a1` < 0.
  - ...`a1` is positive, `a2` is negative, and `a2 - a1` > 0.

- `Jmp dst` jumps to the address at `dst`.
- `Je dst` jumps to the address at `dst` if `ZF` is set.
- `Jne dst` jumps to the address at `dst` if `ZF` is *not* set.
- `Jl dst` jumps to the address at `dst` if `SF` and `OF` have different values.
- `Jg dst` jumps to the address at `dst` if `SF` and `OF` are set to the same value and `ZF` is unset.
- `And dst src` computes the bitwise AND (`&`) of the operands and stores the result in `dst`.
- `Or dst src` computes the bitwise OR (`|`) of the operands and stores the result in `dst`.
- `Xor dst src` computes the bitwise XOR (`^`) of the operands and stores the result in `dst`.
- `Sal dst i` arithmetically shifts the bits in `dst` to the left by `i` bits and stores the result in `dst`. The new bits from the right are `0`s, and the `CF` flag is updated to the value of the most-significant bit during each shift.

  NOTE: When `i` is `1`, the `OF` flag is set if the most-significant bit of the result is different from the value of the `CF` flag, and cleared if they are the same.

- `Sar dst src` arithmetically shifts the bits in `dst` to the right by `i` bits and stores the result in `dst`. The new bits from the left are duplicated from the original most-significant bit, and the `CF` flag is updated to the value of the least-significant bit during each shift.

  NOTE: In contrast to `Sal`, the `OF` flag is always cleared for `Sar`.

- `Push src` decrements the stack pointer and stores the `src` operands on the top of the stack.
- `Pop dst` loads the value from the top of the stack into the `dst` operand and increments the stack pointer.
- `Lea dst l` loads the address of the label `l` and stores it in `dst`.