Cluster

An assembler with a difference

Robert Strandh

2015

Contents

Chapter 1

Introduction

1.1 Purpose

Cluster is an assembler [Sal92], but it differs from traditional assemblers in some crucial ways:

- It does not take a *source file* as its input. Instead, instructions and labels are represented as *standard objects*.[1] This way, we avoid the problem of having to define an input syntax for instructions.
- It does not produce *object code* as output. Instead, it produces a Common Lisp vector of unsigned eight-bit bytes.
- There must not be any unresolved references in the program submitted to Cluster. All references to labels must have a corresponding label defined.

Cluster is mainly meant to server as a *backend* for compilers written in Common Lisp, but they can be compilers for any language. With Cluster, the compiler does not need to turn instructions into surface syntax or S-expressions, only to have the assembler parse that output into some internal representation right

[1] Recall that "a standard object is an instance of (a subclass of) the class named `standard-object`. A standard object is what you obtain when you call `make-instance` of a class defined with `defclass`.

away. By avoiding this pair of unparsing/parsing step, the speed of the compilation process is slightly improved. But the main reason for Cluster to avoid surface syntax is that it can easily become ambiguous over time, as more types of instructions and operands need to be expressed. It is much easier to extend a CLOS class to handle new situations as the arise.

Chapter 2

X86 instruction database

Cluster contains a database with the description of a significant number of instructions. Each instruction is described by one or more occurrences of calls to the macro `define-instruction` described below. A set of calls is recognized as containing different variations on the same instruction by the fact that the required parameter `mnemonic` is the same string for each member of the set.

⇒ `define-instruction` *mnemonic* `&key` *modes operands opcodes opcode-extension encoding lock operand-size-override rex.w* [*Macro*]

2.1 Interpreting Intel instruction reference pages

The instruction reference pages provided by Intel contain complete descriptions of how each instruction and its variants are encoded. This key to understanding this description is given in section 3.1 in the Intel manuals. Here, we give a more direct description of the correspondence between the Intel reference pages and the arguments that should be supplied to the `define-instruction` macro.

2.1.1 Modes

The Intel documentation contains two columns mentioning *modes*. One column says "64-bit Mode" and the other says "Compat/Leg Mode" (which means compatibility or legacy mode). This information is reflected in the keyword argument `:modes` to `define-instruction`. The value of this keyword argument is a list of one or two elements each of which is either the integer 32 or the integer 64. When the Intel documentation mentions "Valid" in the column "64-bit Mode" the list should contain 64. When the column "Compat/Leg Mode" mentions "Valid", then the list should contain 32.

2.1.2 Operands

The `define-instruction` macro has a keyword argument `:operands`. The argument should be a list of possible operands. Valid operands are:

- (gpr-a 8). The lower 8 bits of register A, also referred to as AL.
- (gpr-a 16). The 16-bit register AX.
- (gpr-a 32). The 32-bit register EAX.
- (gpr-a 64). The 64-bit register RAX.
- (gpr 8). Any 8-bit register.
- (gpr 16). Any 16-bit register.
- (gpr 32). Any 32-bit register.
- (gpr 64). Any 64-bit register.
- (imm 8). An 8-bit immediate operand.
- (simm 8). An 8-bit immediate operand, sign extended to the size of the other operand. This operand can occur only as the second of two operands where the first operand has a size that is greater than 8 bits.
- (imm 16). A 16-bit immediate operand.

- (simm 16). A 16-bit immediate operand, sign extended to the size of the other operand. This operand can occur only as the second of two operands where the first operand has a size that is greater than 16 bits.
- (imm 32). A 32-bit immediate operand.
- (simm 32). A 32-bit immediate operand, sign extended to the size of the other operand. This operand can occur only as the second of two operands where the first operand has a size that is greater than 32 bits.
- (imm 64). A 64-bit immediate operand.
- (memory 8). An 8-bit memory operand.
- (memory 16). A 16-bit memory operand.
- (memory 32). A 32-bit memory operand.
- (memory 64). A 64-bit memory operand.

2.1.3 Opcodes

The `define-instruction` macro has a keyword argument `:opcodes`. This argument should be a list of unsigned octets. It is preferable to use hexadecimal notation for the opcodes. The Intel instruction reference pages contain a column labeled either "Opcode" or "Opcode/Instruction". The opcodes are indicated in that column as a sequence of hexadecimal values, sometimes followed by "/n" (where n is a small non-negative integer) or "/r". The occurrence of "/r" does not need to be encoded in the `define-instruction` form.

Sometimes, the list of opcodes is preceded by "REX.W+". When that is the case, the `:rex.w` keyword argument should be provided, with a value of `t`.

2.1.4 Encoding

The `:encoding` keyword argument to `define-instruction` is a list of the same length as the one supplied to `:operands`. Each element of the list is an encoding. For each operand, the encoding indicates how that operand is

encoded in the binary version of the instruction. Possible values for an encoding are:

- `modrm`. This value indicates that the operand is encoded in the "mod" and "r/m" fields of the instruction. It should be used when the corresponding entry in the Intel documentation indicates r/m8, r/m16, r/m32, or r/m64.
- `reg`. This value indicates that the operand is encoded in the "reg" field of the instruction. It should be used when the corresponding entry in the Intel documentation indicates r8, r16, r32, or r64.
- `imm`. This value indicates that the operand is encoded as an immediate value in the instruction stream. It should be used when the corresponding operand in the `:operands` keyword argument mentions `(imm 8)`, `(imm 16)`, `(imm 32)`, or `(imm 64)`.
- `label`. This value indicates that the corresponding operand is the target of a control-transfer instruction. It should be used when the corresponding operand in the `:operands` keyword argument mentions `(label 8)`, `(label 16)`, `(label 32)`, or `(label 64)`.
- `-`. This value indicates that the corresponding operand is implicitly defined by the opcode, which is often the case when general-purpose register A is one of the operands of the instruction.
- `+r`. This value use used in some cases when the corresponding operand is a register. It means that some small integer that indicates a register is added to the opcode itself. This value should be used when the Intel documentation has a +rb, +rw, or +rd in the opcode column.

2.1.5 Operand-size override

The keyword argument `:operand-size-override` to `define-instruction` is required to have a true value when the particular operand requires an instruction prefix in order for that size operand to be used. In 32-bit and 63-bit mode, the default operand size is 32-bits, so for instance if the `push` instruction is to be given a 16-bit value instead of a 32-bit value in one of these modes, then the operand-size-override prefix must be given. For this prefix to be emitted, this keyword must be supplied with a true value.

2.1.6 Atomic execution

When the instruction supports atomic execution, the keyword `:lock` with a true argument should be given to `define-instruction`. This value then indicates that the instruction is capable of executing atomically.

2.1.7 64-bit prefix

In order to access more than 8 registers, Intel instructions require a prefix called REX.W. When the Intel documentation mentions this prefix in the opcode column, the keyword argument `rex.w` whould be supplied to `define-instruction` with a true value.

Chapter 3

Future additions to this library

Bibliography

[Sal92] David Salomon. *Assemblers and Loaders.* Ellis Horwood, Upper Saddle River, NJ, USA, 1992.