A Finite State Machine to Determine the Starting Byte Opcodes of Instructions Encoded in IA-32

1 Introduction

A parallel prefix circuit can, in theory, determine the position of starting byte opcodes in logarithmic time from a stream of instructions encoded in a complex instruction set architecture (CISA)[1]. A complex instruction (such Intel's IA-32 architecture) is not fixed length and the length of the instruction may not be fully indicated by the starting byte opcode. This poses a significant challenge to the decoding circuit. Typically, to find the starting byte of instructions, a microprocessor steps through the instructions in linear time. A parallel prefix circuit, however, may be able to find these same byte positions in logarithmic time, no matter how complex the encoding. The first step to designing this circuit is to design the finite state machine (FSM) for the instruction set architecture.

Presented is a FSM to decode Intel's IA-32 "x86" instruction set architecture. The FSM steps through the instructions, much as a microprocessor would, and determines the position and value of the starting byte opcode in linear time. The instruction set is assumed to be running on a machine with a 32 bit word length.

2 Description of Instruction Set

Intel's IA-32, x86 architecture consists of a non fixed-length instruction set. Additionally, the primary byte opcode does not fully specify the length of the instruction. The instruction may consist of opcodes, ModR/M bytes, SIB bytes, displacement bytes, and immediate bytes. The ModR/M byte is an "addressing-form specifier byte" and is used to refer to operands in memory or registers. Additionally bits 3,4 and 5 may act as opcode extension bits. A SIB byte may be used with the ModR/M to indicate an additional scale, index, or base.

As described in the Instruction Set Reference [2], a complete instruction consists of up to 4 optional instruction prefixes, a mandatory one or two byte opcode, an optional Mod R/M byte, an optional SIB byte, optional displacement bytes and optional immediate date bytes. The presence of the Mod R/M is specified by the opcode, the SIB byte by the Mod R/M byte, the displacement bytes by the opcode, the Mod R/M, and/or the SIB byte, and the immediate bytes are specified by the opcode.

The opcode byte may specify a ModR/M byte, 1 or 4 bytes of displacement (2 bytes are possible on a machine with a 16 bit word size), and 1 or 4 bytes of immediate data (again, 2 bytes are possible). When the primary byte opcode is 0x0F or 0xD8 through 0xDF, a second byte of opcode follows. This second byte will specify the encoding of the instruction.

The Mod R/M byte may specify a SIB byte, and/or 1 or 4 bytes of additional displacement (2 bytes are also possible). The Mod R/M encoding falls into one of 8 categories, indicated by Table 1. Additionally, bits 3,4 and 5 of the ModR/M byte may serve as an opcode extension. This extension may change the encoding specified by the primary byte opcode. For instance, when the primary byte opcode 0xF7 is followed by a ModR/M whose 3rd, 4th and 5th bytes are 000 respectively, a TEST instruction is indicated and 4 bytes of immediate follow the ModR/M encoding. If the 3rd, 4th, and 5th bytes are 010, than a NOT instruction is specified and there will be no bytes following the ModR/M encoding.

Table 1 : Mod R/M encoding

	Col1	Col2	Col3	Col4	Col5	Col6	Col7	Col8
ModR/M values	00-03 06-0B 0E-13 16-1B 1E-23 26-2B 2E-33 36-3B 3E-3F	04 0C 14 1C 24 2C 34 3C	40-43 45-4B 4D-53 55-5B 5D-63 65-6B 6D-73 67-7B 7D-7F	44 4C 54 5C 64 6C 74 7C	80-83 85-83 8D-93 95-9B 9D-A3 A5-AB AD-B3 B5-BB BD-BF	84 8C 94 9C A4 AC B4 BC	C0-FF	05 0D 15 1D 25 2D 35 3D
opcode encoding without additional disps and imms specified by opcode	opcode | ModR/M	opcode | ModR/M | SIB* | Disp | Disp | Disp | Disp	opcode | ModR/M | Disp	opcode | ModR/M | SIB | Disp	opcode | ModR/M | Disp | Disp | Disp | Disp	opcode | ModR/M | Sib | Disp | Disp | Disp | Disp	opcode | ModR/M	opcode | ModR/M | Disp | Disp | Disp | Disp

* Only certain values of SIB lead to displacement. These occur when ((SIB | 0xF8) == 0xFD)

3 Implementation

The FSM capable of decoding the described instruction set architecture is shown in Figure 1. Appendix 1 contains the C code that implements this FSM. Contrary to the description in the ISA, all prefixes, except 0x66 are labeled as primary byte opcodes. The bytes following them are also labeled as primary byte opcodes.

The code in Appendix 1 generates the file fsmoutfile which indicates the position and value of each of the starting byte opcodes. Currently, the FSM only recognizes starting byte opcodes that were encountered during testing. If the FSM encounters an opcode it does not recognize, it will print an error message and exit. To continue, the opcode will then need to be added to one of the opcode categories described in the Appendix. These are hasmodrm0, hasmodrm1, hasmodrm4, has1add, has2add, has3add, has4add, has6add, and onebyte. The hasmodrm<n> are opcodes that specify a ModR/M and n bytes of displacement or immediate following. has<n>add are opcodes which have n bytes of displacement or immediate following the opcode (no ModR/M). onebyte are opcodes that do not specify any further bytes.

There are several opcode encodings that are not implemented by this FSM. These include unary group 7, a set of instructions that have different specifications based on the Mod R/M extension bits. Additionally, group 3 (primary byte opcode 0xF7 as described above) is not fully implemented. Also, if the ISA were extended to include a 16 bit word-size, the 16 bit ModR/M needs to be encoded.

4 Discussion

The FSM as specified can not accurately decode all given input. Errors may be undetectable, and irrecoverable. For instance, if the compiler generates a bad series of opcodes that the FSM recognizes as a valid set, the FSM may start labeling primary byte opcodes improperly. The instruction fetch of a machine may avoid these instructions depending on control flow. Possibly, a piece of hand compiled assembly may take advantage of control flow by making different instructions within the same subset. Since the FSM is not aware of control flow, it is impossible for it to fully handle these situations.

References

[1] Dana S. Henry and Bradley C. Kuszmaul. Efficient Circuits for Out-of-Order Microprocessors. *October 1998 Disclosure.*

[2] Volume 2: Instruction Set Reference. Intel Architecture Software Developer's Manual. 1997.

Figure 1: FSM to decode IA-32 architecture