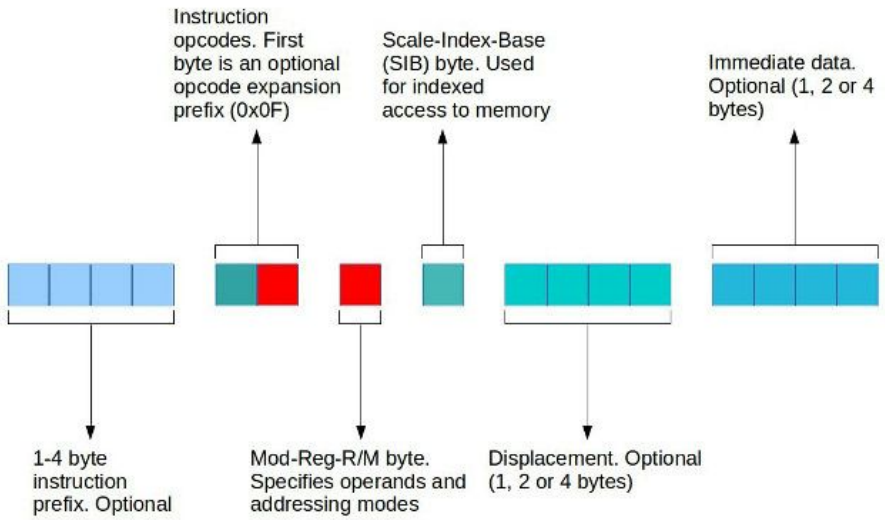
X86 Instruction Decoder

CS 505 - Computer Structure
Ashish Jindal (aj523)

Instruction encoding -



x86 instruction encoding

My Design -

Processing the variable length Instructions -

The most complicated part of the design was to figure out the instruction size and accordingly read next instruction from memory or decode the current instruction. To handle this I implemented a state machine based design. I kept a register large enough to accommodate the largest instruction inside the decoder and named it “decode_reg” and kept another register of same size called “decode_hold_reg” which will hold the residual data from decode_reg.

For e.g. if an instruction is of size 2 bytes but decode_reg read 4 bytes instead, so at the end of decoding process, it will shift the remaining 2 bytes to the hold register, which are later on appended to the next 4 byte value read into the decode register. This way I can process the flexible instruction size.

Mod/Reg/RM decoding -

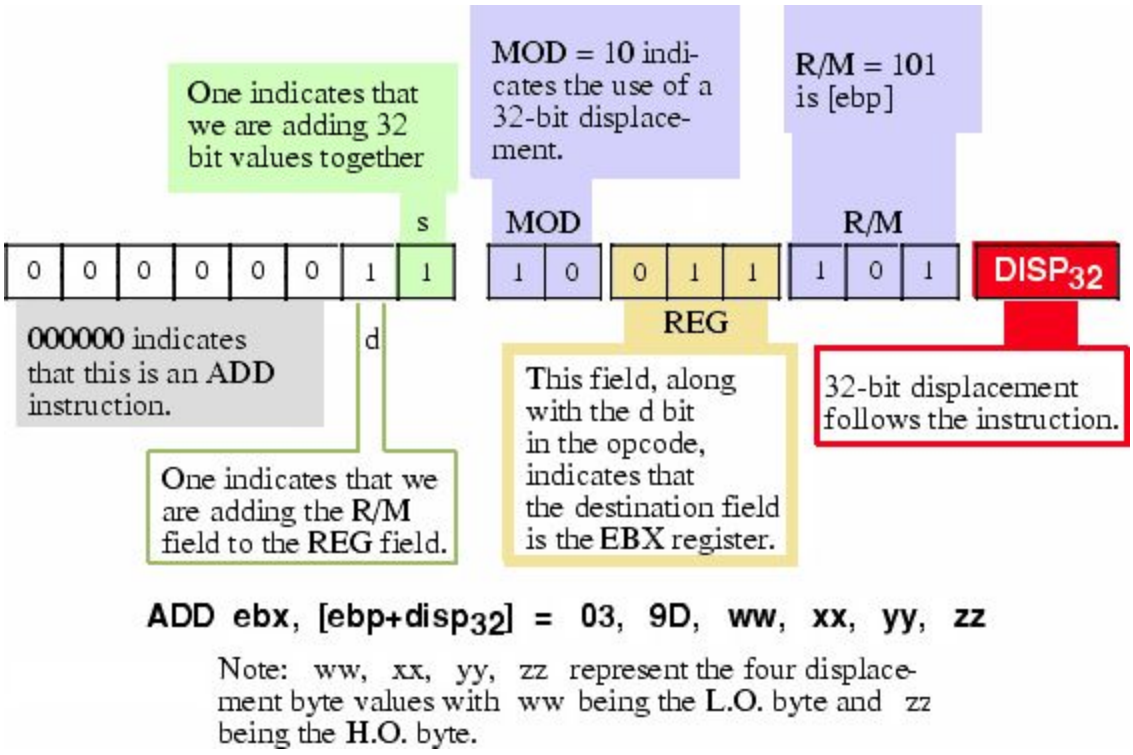
I Implemented a separate function called “decode_mod_reg_rm()” which will independently handle the task of decoding the mod/reg/rm byte for the instructions in a generic way. All the

addressing modes are supported in this function accept the immediate one. So if an instruction wants to do immediate addressing, it can only happen if the mod bits point to direct addressing mode. Currently I haven't copied this implementation to other addressing modes. Apart from this I have separated the direction specifying bit from opcode, but haven't used it while decode mod/reg/rm to change the src/dest directions.

Registers -

Instructions can require access to 8/16/32 bit registers. 8 or 32 bits register mode bits are embedded in the least significant bit of the opcode and for 16 bit register mode we need to specify a separate prefix "66" in prefix byte. In my code the required registers are handled in function "get_reg()"

Example encoding supported by my implementation -



***Sample input (Also demonstrating above described decoding for add instruction) can be found in the sample file named - sample_input.binhex, the corresponding output is in file sample_output.**

Instructions supported by my implementation -

1. Add
2. Sub
3. And
4. Cmp
5. Xor
6. Xchg
7. Pop
8. Push
9. Dec
10. Div
11. Mov
12. Lea
13. Call
14. Jmp
15. ret
16. hlt

My mod-reg-rm decoding is quite generic and it works for all instruction that require 2 or more operands very well. For single operand or no operand instruction the decoding is buggy/incomplete. Also it handles the immediate addressing mode only for direct addressing (Mod bits = 2'b11) for now.
All the instructions requiring 2 or more operands should work correctly with this implementation, but I could only plug in opcodes for above mentioned ones because of time constraint.
Apart from this my implementation handles only 1-byte opcodes. Because of incomplete implementation it can't be run on objcopy output of c programs as it won't be able to identify many instructions. So I tested specific instructions by creating my own hex files.

Resources -

1. Intel x86 manual.
2. Book - Art of assembly language.
3. Codeproject tutorial - Link