Go 1.11 release introduces AVX-512 support.
This page describes how to use new features as well as some important encoder details.

Terminology

Most terminology comes from Intel Software Developer's manual.
Suffixes originate from Go assembler syntax, which is close to AT&T, which also uses size suffixes.

Some terms are listed to avoid ambiguity (for example, opcode can have different meanings).

Term	Description
Operand	Same as "instruction argument".
Opcode	Name that refers to instruction group. For example, `VADDPD` is an opcode. It refers to both VEX and EVEX encoded forms and all operand combinations. Most Go assembler opcodes for AVX-512 match Intel manual entries, with exceptions for cases where additional size suffix is used (e.g. `VCVTTPD2DQY` is `VCVTTPD2DQ`).
Opcode suffix	Suffix that overrides some opcode properties. Listed after "." (dot). For example, `VADDPD.Z` has "Z" opcode suffix. There can be multiple dot-separated opcode suffixes.
Size suffix	Suffix that specifies instruction operand size if it can't be inferred from operands alone. For example, `VCVTSS2USIL` has "L" size suffix.
Opmask	Used for both `{k1}` notation and to describe instructions that have `K` registers operands. Related to masking support in EVEX prefix.
Register block	Multi-source operand that encodes register range. Intel manual uses `+n` notation for register blocks. For example, `+3` is a register block of 4 registers.
FP	Floating-point

New registers

EVEX-enabled instructions can access additional 16 `X` (128-bit xmm) and `Y` (256-bit ymm) registers, plus 32 new `Z` (512-bit zmm) registers in 64-bit mode. 32-bit mode only gets `Z0-Z7`.

New opmask registers are named `K0-K7`.
They can be used for both masking and for special opmask instructions (like `KADDB`).

Masking support

Instructions that support masking can omit `K` register operand.
In this case, `K0` register is implied ("all ones") and merging-masking is performed.
This is effectively "no masking".

`K1-K7` registers can be used to override default opmask.
`K` register should be placed right before destination operand.

Zeroing-masking can be activated with `Z` opcode suffix.

For example, `VADDPD.Z (AX), Z30, K3, Z10` uses zeroing-masking and explicit `K` register.

- If `Z` opcode suffix is removed, it's merging-masking.
- If `K3` operand is removed, `K0` operand is implied.

It's compile-time error to use `K0` register for `{k1}` operands (consult manuals for details).

EVEX broadcast/rounding/SAE support

Embedded broadcast, rounding and SAE activated through opcode suffixes.

For reg-reg FP instructions with `{er}` enabled, rounding opcode suffix can be specified:

- `RU_SAE` to round towards +Inf
- `RD_SAE` to round towards -Inf
- `RZ_SAE` to round towards zero
- `RN_SAE` to round towards nearest

> *To read more about rounding modes, see MXCSR.RC info.*

For reg-reg FP instructions with `{sae}` enabled, exception suppression can be specified with `SAE` opcode suffix.

For reg-mem instrictons with `m32bcst/m64bcst` operand, broadcasting can be turned on with `BCST` opcode suffix.

Zeroing opcode suffix can be combined with any of these.
For example, `VMAXPD.SAE.Z Z3, Z2, Z1` uses both `Z` and `SAE` opcode suffixes.
It is important to put zeroing opcode suffix last, otherwise it is a compilation error.

Register block (multi-source) operands

Register blocks are specified using register range syntax.

It would be enough to specify just first (low) register, but Go assembler requires explicit range with both ends for readability reasons.

For example, instructions with `+3` range can be used like `VP4DPWSSD Z25, [Z0-Z3], (AX)`.
Range `[Z0-Z3]` reads like "register block of Z0, Z1, Z2, Z3".
Invalid ranges result in compilation error.

AVX1 and AVX2 instructions with EVEX prefix

Previously existed opcodes that can be encoded using EVEX prefix now can access AVX-512 features like wider register file, zeroing/merging masking, etc. For example, `VADDPD` can now use 512-bit vector registers.

See encoder details for more info.

Supported extensions

Best way to get up-to-date list of supported extensions is to do `ls -1` inside test suite directory.

Latest list includes:

```
aes_avx512f
avx512_4fmaps
avx512_4vnniw
```

```
avx512_bitalg
avx512_ifma
avx512_vbmi
avx512_vbmi2
avx512_vnni
avx512_vpopcntdq
avx512bw
avx512cd
avx512dq
avx512er
avx512f
avx512pf
gfni_avx512f
vpclmulqdq_avx512f
```

128-bit and 256-bit instructions additionally require `avx512vl`.
That is, if `VADDPD` is available in `avx512f`, you can't use `X` and `Y` arguments without `avx512vl`.

Filenames follow `GNU as` (gas) conventions.
avx512extmap.csv can make naming scheme more apparent.

Instructions with size suffix

Some opcodes do not match Intel manual entries.
This section is provided for search convenience.

Intel opcode	Go assembler opcodes
`VCVTPD2DQ`	`VCVTPD2DQX`, `VCVTPD2DQY`
`VCVTPD2PS`	`VCVTPD2PSX`, `VCVTPD2PSY`
`VCVTTPD2DQ`	`VCVTTPD2DQX`, `VCVTTPD2DQY`
`VCVTQQ2PS`	`VCVTQQ2PSX`, `VCVTQQ2PSY`
`VCVTUQQ2PS`	`VCVTUQQ2PSX`, `VCVTUQQ2PSY`
`VCVTPD2UDQ`	`VCVTPD2UDQX`, `VCVTPD2UDQY`
`VCVTTPD2UDQ`	`VCVTTPD2UDQX`, `VCVTTPD2UDQY`
`VFPCLASSPD`	`VFPCLASSPDX`, `VFPCLASSPDY`, `VFPCLASSPDZ`
`VFPCLASSPS`	`VFPCLASSPSX`, `VFPCLASSPSY`, `VFPCLASSPSZ`
`VCVTSD2SI`	`VCVTSD2SI`, `VCVTSD2SIQ`
`VCVTTSD2SI`	`VCVTSD2SI`, `VCVTSD2SIQ`
`VCVTTSS2SI`	`VCVTSD2SI`, `VCVTSD2SIQ`
`VCVTSS2SI`	`VCVTSD2SI`, `VCVTSD2SIQ`
`VCVTSD2USI`	`VCVTSD2USIL`, `VCVTSD2USIQ`
`VCVTSS2USI`	`VCVTSS2USIL`, `VCVTSS2USIQ`

`VCVTTSD2USI`	`VCVTTSD2USIL`, `VCVTTSD2USIQ`
`VCVTTSS2USI`	`VCVTTSS2USIL`, `VCVTTSS2USIQ`
`VCVTUSI2SD`	`VCVTUSI2SDL`, `VCVTUSI2SDQ`
`VCVTUSI2SS`	`VCVTUSI2SSL`, `VCVTUSI2SSQ`
`VCVTSI2SD`	`VCVTSI2SDL`, `VCVTSI2SDQ`
`VCVTSI2SS`	`VCVTSI2SSL`, `VCVTSI2SSQ`
`ANDN`	`ANDNL`, `ANDNQ`
`BEXTR`	`BEXTRL`, `BEXTRQ`
`BLSI`	`BLSIL`, `BLSIQ`
`BLSMSK`	`BLSMSKL`, `BLSMSKQ`
`BLSR`	`BLSRL`, `BLSRQ`
`BZHI`	`BZHIL`, `BZHIQ`
`MULX`	`MULXL`, `MULXQ`
`PDEP`	`PDEPL`, `PDEPQ`
`PEXT`	`PEXTL`, `PEXTQ`
`RORX`	`RORXL`, `RORXQ`
`SARX`	`SARXL`, `SARXQ`
`SHLX`	`SHLXL`, `SHLXQ`
`SHRX`	`SHRXL`, `SHRXQ`

Encoder details

Bitwise comparison with older encoder may fail for VEX-encoded instructions due to slightly different encoder tables order.

This difference may arise for instructions with both `{reg, reg/mem}` and `{reg/mem, reg}` forms for reg-reg case. One of such instructions is `VMOVUPS`.

This does not affect code behavior, nor makes it bigger/less efficient.
New encoding selection scheme is borrowed from Intel XED.

EVEX encoding is used when any of the following is true:

- Instruction uses new registers (High 16 `X` / `Y`, `Z` or `K` registers)
- Instruction uses EVEX-related opcode suffixes like `BCST`
- Instruction uses operands combination that is only available for AVX-512

In all other cases VEX encoding is used.
This means that VEX is used whenever possible, and EVEX whenever required.

Compressed disp8 is applied whenever possible for EVEX-encoded instructions.
This also covers broadcasting disp8 which sometimes has different N multiplier.

Experienced readers can inspect avx_optabs.go to learn about N multipliers for any instruction.

For example, `VADDPD` has these:

- `N=64` for 512-bit form; `N=8` when broadcasting
- `N=32` for 256-bit form; `N=8` when broadcasting
- `N=16` for 128-bit form; `N=8` when broadcasting

Examples

Exhaustive amount of examples can be found in Go assembler test suite.

Each file provides several examples for every supported instruction form in particular AVX-512 extension.
Every example also includes generated machine code.

Here is adopted "Vectorized Histogram Update Using AVX-512CD" from Intel® Optimization Manual:

```
for i := 0; i < 512; i++ {
  histo[key[i]] += 1
}
```

```
top:
        VMOVUPS      0x40(SP)(DX*4), Z4  //; vmovups      zmm4, [rsp+rdx*4+0x40]
        VPXORD       Z1, Z1, Z1          //; vpxord       zmm1, zmm1, zmm1
        KMOVW        K1, K2              //; kmovw        k2, k1
        VPCONFLICTD Z4, Z2               //; vpconflictd zmm2, zmm4
        VPGATHERDD   (AX)(Z4*4), K2, Z1  //; vpgatherdd   zmm1{k2}, [rax+zmm4*4]
        VPTESTMD     histo<>(SB), Z2, K0 //; vptestmd     k0, zmm2, [rip+0x185c]
        KMOVW        K0, CX              //; kmovw        ecx, k0
        VPADDD       Z0, Z1, Z3          //; vpaddd       zmm3, zmm1, zmm0
        TESTL        CX, CX              //; test         ecx, ecx
        JZ           noConflicts         //; jz           noConflicts
        VMOVUPS      histo<>(SB), Z1     //; vmovups      zmm1, [rip+0x1884]
        VPTESTMD     histo<>(SB), Z2, K0 //; vptestmd     k0, zmm2, [rip+0x18ba]
        VPLZCNTD     Z2, Z5              //; vplzcntd     zmm5, zmm2
        XORB         BX, BX              //; xor          bl, bl
        KMOVW        K0, CX              //; kmovw        ecx, k0
        VPSUBD       Z5, Z1, Z1          //; vpsubd       zmm1, zmm1, zmm5
        VPSUBD       Z5, Z1, Z1          //; vpsubd       zmm1, zmm1, zmm5

resolveConflicts:
        VPBROADCASTD CX, Z5     //; vpbroadcastd zmm5, ecx
        KMOVW CX, K2            //; kmovw        k2, ecx
        VPERMD Z3, Z1, K2, Z3   //; vpermd       zmm3{k2}, zmm1, zmm3
        VPADDD Z0, Z3, K2, Z3   //; vpaddd       zmm3{k2}, zmm3, zmm0
        VPTESTMD Z2, Z5, K2, K0 //; vptestmd     k0{k2}, zmm5, zmm2
        KMOVW K0, SI            //; kmovw        esi, k0
        ANDL SI, CX             //; and          ecx, esi
        JZ noConflicts          //; jz           noConflicts
        ADDB $1, BX             //; add          bl, 0x1
        CMPB BX, $16            //; cmp          bl, 0x10
        JB resolveConflicts     //; jb           resolveConflicts

noConflicts:
        KMOVW       K1, K2             //; kmovw       k2, k1
        VPSCATTERDD Z3, K2, (AX)(Z4*4) //; vpscatterdd [rax+zmm4*4]{k2}, zmm3
        ADDL        $16, DX            //; add         edx, 0x10
        CMPL        DX, $1024          //; cmp         edx, 0x400
        JB          top                //; jb          top
```