Using SIMD in/out instructions

Ross J. Maloney

November 2021

Abstract

To write code to exploit the potential of SIMD hardware, knowledge beyond existence of SIMD on a computer is needed. Movement of five computational data types between memory and the SIMD registers, and between the SIMD registers, is explored. Relative data transfer execution times for xmm, ymm, and zmm registers are tabulated from experimental measurements and compared with standard register transfer times. Specifying data moving from aligned and unaligned memory boundaries are shown to have consequences.

There are two types SIMD instruction which transfer data between SIMD registers xmm, ymm and zmm and memory. These are aligned and unaligned. The alignment is associated with the data supplied to the instruction. In the case of aligned, the behaviour can change with each run of the containing program. If an aligned instruction is used, but the data is not correctly aligned, then an exception is thrown, and a `Segmentation Fault` results. If an unaligned instruction is used, no exception is thrown when using aligned or unaligned data. Conclusion: Always use unaligned instructions. This would simplify the writing of the code which uses these SIMD instructions. But is there a cost resulting from this use? Experiments were performed to find out.

The SIMD instruction reference AMD (2020) provides complete information on all the 128-bit and 256-bit handling SIMD assembler instructions, but not the 512-bit handling instructions. By contrast, ORACLE (2020) covers all three bit widths but contains less information on each instruction. Aligned and unaligned data referencing instructions are covered, but the programming consequence of using either is not elaborated upon. Are all such instructions equal?

1 Experimental approach

Experiments were performed using a purpose written computer program. The mainline called a procedure which performed the SIMD interaction of interest. The mainline called this procedure a number of times, and the time taken to perform those calls was recorded. Figure 1 shows an example of the code used. In performing the experiments, the mainline and the procedure were in separate files; one for the C language mainline and the other for the assembler prceedure. The assembler file was processed using command `nasm -f elf64` and `clang` was used to compile the C mainline and link it with the object file produced by `nasm`. All experiments were performed on a MacPro 2019 running Debian SID Bullseye Linux with X Window running using a single core. Each experiment was run from a `st` text window with a dormant `vim` editor in another text window. There were no other user processes executing.

To obtain timing measures of a reasonable duration 5,000,000,000 repeated calls of the assembler procedure were measured by the mainline.

The codes in Figure 1 and Figure 3 show procedure `my_get_wtime()` used to measure the execution time. This procedure returned the time of day at the instance of it's call; the difference in such times between the second and first call was used as the measure of execution time.

Each combination of variables encompassed in the software were executed 25 times. From the timing measurements produced, the maximum and minimum execution times was recorded. The mean of those 25 timings was also recorded.

Two sets of experiments were performed. The first moved data into and out of a prescribed SIMD registers. Each assembler procedure call executed one movement of data from memory into the SIMD register and then moved that data from the SIMD register into memory. For these experiments it was assumed loading and unloading the SIMD registers took the same length of time. The second set moved data between SID registers. One assembler call was used to load a prescribed SIMD register with data. This loading was not timed. Then two SIMD instructions contained in another assembler procedure were used to move this data into two separate SIMD registers. This movement was timed.

In each set of experiments the behaviour of data movement used `xmm`, `ymm` and `zmm` SIMD registers representing 128, 256, and 512 bit data clusters respectively, were considered. The register type used together with the register number were part of the operand of the instruction contained in the assembler procedure. The effect of forming clusters from 32 and 64 bit floating point values, then 16, 32, and 64 bit integer values were covered in the experiments.

A number of assumptions were made. It was assumed the data transfer rate between SIMD registers of the same type was the same for all registers of that type present, for example register type `ymm` number 0 had the same transfer rate as `ymm` number 5. Another assumption was using an unaligned instruction to load data from memory into a register had no influence on the behaviour of using aligned or unaligned instructions to move data between such registers. Further it was assumed transfer rates were the same for all cores on a multi-core processor chip and thus no reference to a particular core use was needed in the experiment code.

1.1 Between memory and SIMD registers

Operation of SIMD processing requires movement of data into the SIMD registers, processing of that data in those registers, and moving the resulting data back into memory. Moving data into and out from those SIMD registers was of interest here.

As would be expected there are assembler instructions to move data into and out of memory and the SIMD registers. The same instruction mnemonic is used to move the data in and out, the direction of movement determined by the placement order of the operands.

Different instructions are used when the memory data is aligned or unaligned on specified memory boundaries. However, the aligned or unaligned mnemonics are discriminated by a `a` or `u` respectively in in the instruction mnemonic, for example the aligned instruction `vmovaps` and the unaligned instruction `vmovups`.

Figure 1 shows an example of the coding used to derive one of the data combinations collectively shown in Table 1.

With respect to Table 1 the following convention was followed. In the `instruction` and the `data type` columns, a black entry means no change, the non-blank entry above in the column applies to this entry. In the `faults?` column, a blank entry means `no`, or no faults occurred, for this entry.

An important note in Table 1 data is the occurrence of faults. These faults were `Segmentation`

faults. They occurred when using ymm and zmm registers in instructions with aligned data , but not with xmm registers. When floating point data was being transferred, repeated attempts to run the experiment would oscillate between Segmentation fault and running to completion there by producing an execution time measure. With integer data Segmentation faults were only produced with zmm registers.

```
#include  <stdio.h>
#include  <time.h>

extern void simd_io(float[], float[]);

int  main()
{
  long  i;
  double  tick;
  double  my_get_wtime(void);
  float  x[] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.0, 10.10,
     11.11, 12.12, 13.13, 14.14, 15.15, 16.16, 17.17};
  float  z[] = {45.4, 12.9, -90.4, 12.0, -3.0, 67.9, 56.3, 10.9,
     -12.5, 7.4, -65.2, 12.3, 43.7, 12.0, 34.0, 15.9, -8.9};

  tick = my_get_wtime();
  for (i = 0; i < 5000000000; i++)  simd_io(x, z);
  tick = my_get_wtime() - tick;
  printf("tick = %lf\n", tick);

  return(0);
}

double my_get_wtime()
{
  struct timespec  ts;
  double           time, nano;

  clock_gettime(CLOCK_MONOTONIC, &ts);
  time = (double) ts.tv_sec;
  nano = (double) ts.tv_nsec;
  return (time + nano/1000000000.0);
}

;  read data from memory into a SIMD register.
;  void simd_io(float[], float[])

      global  simd_io

      section .text

simd_io:
      vmovups   ymm0, [rsi]    ; put the data into SIMD register
      vmovups   [rdi], ymm0    ; retrieve data from SIMD register
      ret
```

Figure 1: C and assembler code used to measure SIMD/memory speed

Table 1: Timing results of moving data between memory and xmm, ymm, and zmm registers

				Execution time [sec]		
instructions	registers	data type	faults?	min	mean	max
vmovaps	xmm	32-bit float		7.16	7.27	7.38
	ymm		Yes	7.09	7.26	7.38
	zmm		Yes	8.11	8.43	8.54
vmovups	xmm			6.97	7.23	7.34
	ymm			6.90	7.02	7.10
	zmm			7.99	8.44	8.69
vmovapd	xmm	64-bit float		6.94	6.98	7.03
	ymm		Yes	6.93	6.96	6.99
	zmm		Yes	8.02	8.05	8.14
vmovupd	xmm			6.92	6.97	7.02
	ymm			6.92	6.95	7.04
	zmm			8.00	8.04	8.07
vmovdqa	xmm	16-bit integer		5.80	5.83	5.89
	ymm		Yes	5.81	5.84	5.94
vmovdqa32	zmm		Yes			
vmovdqu	xmm			5.79	5.85	5.96
	ymm			5.82	5.94	6.21
vmovdqu32	zmm			8.07	8.49	8.75
vmovdqa	xmm	32-bit integer		7.02	7.26	7.37
	ymm		Yes	6.96	7.16	7.37
vmovdqa32	zmm		Yes			
vmovdqu	xmm			6.94	7.23	7.38
	ymm			6.97	7.24	7.54
vmovdqu32	zmm			8.00	8.04	8.09
vmovdqa	xmm	64-bit integer		6.96	6.99	7.05
	ymm		Yes	6.90	6.94	7.00
vmovdqa32	zmm		Yes			
vmovdqu	xmm			6.94	6.98	7.00
	ymm			6.92	6.96	7.02
vmovdqu32	zmm			8.00	8.05	8.10

Figure 2 shows a plot of some of the Table 1 data to indicate the affect of register used, data type, and data memory alignment on relative performance as indicated by the execution time mean. No memory aligned data using `zmm` registers is shown due to fault occurrence. The colour of the lines group register type combined with both aligned and unaligned memory alignment.

The data of Table 1 and the relationship information of Figure 2 suggests:

- unaligned and aligned data transfers occur at approximately the same transfer rate
- transfers involving `zmm` registers are slower than those using `xmm` and `ymm` registers which is consistent with transferring more data
- transfers involving `xmm` and `ymm` registers occur at approximately the same transfer rate
- using aligned transfers can cause faults

These observations were used in formulating the following set of experiments.

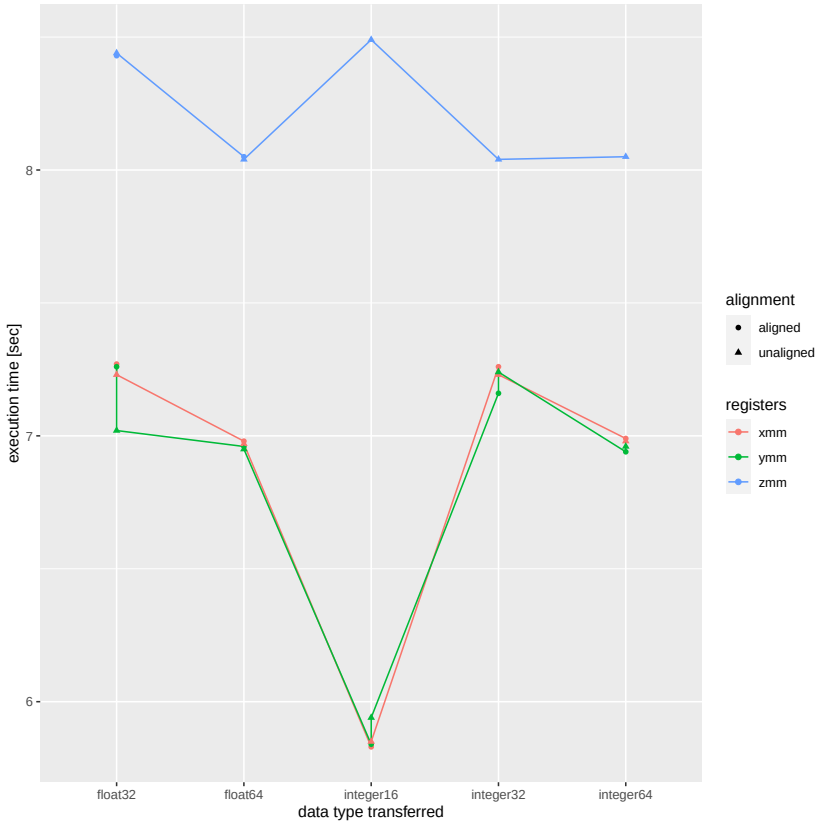

Figure 2: Relationships between SIMD memory and register transfer data

1.2 Between SIMD registers

There are 16 `xmm` and `ymm` registers and 32 `zmm` registers. In the previous experiments moving data into one register and memory was considered. But what is the relative cost of moving data between corresponding register types as opposed to moving register data to and from memory? By moving data in this manner can advantage be obtained from such multiple SIMD registers? The answer to this question would expected to be yes, but evidence is needed. If yes, then handling of SIMD registers should follow the same philosophy of *minimizing data movement into and out of registers* as occurs in standard CPU assembler programming?

Figure 3 shows a sample of the code used to measure execution times of register to register data transfer. Data was first loaded into a SIMD register using the `simd_io` procedure. The SIMD instruction used in procedure `simd_io` was not changed to match the data transfer between registers being tested. To avoid generating faults, the instruction used was of the memory unaligned type. The other correspondence between such instructions and data types as shown in Table 1 were followed. This loading of data from memory was not included in the experiment time results of Table 2.

Each experiment in this set was performed by modifying the code of Figure 3. Procedure call `simd_io` was used to load register 0. The procedure `between` was called repeatedly to copy this data to two other registers. For each experiment the prototype of the `simd_io()` call in the mainline C program was changed to match the data type being transferred. This match was to the data type stored in the mainline. Also, the register type in the two assembler procedures was changed to that being tested. The code combination was then run 25 times.

```
#include  <stdio.h>
#include  <time.h>

extern void simd_io(float[]);
extern void between(void);

int  main()
{
  long  i;
  double  tick;
  double  my_get_wtime(void);
  float  x[] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.0,
     10.10, 11.11, 12.12, 13.13, 14.14, 15.15, 16.16, 17.17};

  simd_io(x);
  tick = my_get_wtime();
  for (i = 0; i < 5000000000; i++)  between();
  tick = my_get_wtime() - tick;
  printf("tick = %lf\n", tick);

  return(0);
}

double my_get_wtime()
{
  struct timespec  ts;
  double           time, nano;

  clock_gettime(CLOCK_MONOTONIC, &ts);
  time = (double) ts.tv_sec;
  nano = (double) ts.tv_nsec;
  return (time + nano/1000000000.0);
}

; moving data into and between simd registers

      global  simd_io
      global  between

      section .text

simd_io:
      vmovups   xmm0, [rdi]    ; put data into register xmm0
      ret

between:
      vmovaps   xmm3, xmm0     ; move data from xmm0 to xmm3
      vmovaps   xmm5, xmm0     ; move data from xmm0 to xmm5
      ret
```

Figure 3: C and assembler code to measure between SIMD register speed

Table 2 contains the results of the experiments. As with Table 1 pertaining to the previous set of experiments, a blank entry in a column means the non-black entry above it in the column applies

also to this entry.

Table 2: Timing results of moving data between xmm, ymm, and zmm registers

instructions	registers	data type	Execution time [sec] min	mean	max
vmovaps	xmm	32-bit float	5.78	5.82	5.94
	ymm		5.79	5.85	5.92
	zmm		7.94	7.96	7.97
vmovups	xmm		5.77	5.79	5.91
	ymm		5.78	5.85	5.92
	zmm		7.95	7.96	7.97
vmovapd	xmm	64-bit float	5.78	5.83	5.93
	ymm		5.79	5.87	5.97
	zmm		6.76	6.80	6.93
vmovupd	xmm		5.77	5.83	5.94
	ymm		5.78	5.87	5.95
	zmm		6.75	6.79	6.91
vmovdqa	xmm	16-bit integer	5.78	5.83	5.93
	ymm		5.79	5.85	5.93
vmovdqa32	zmm		6.75	6.79	6.88
vmovdqu	xmm		5.78	5.83	5.91
	ymm		5.78	5.84	5.95
vmovdqu32	zmm		6.75	6.79	6.86
vmovdqa	xmm	32-bit integer	6.14	6.35	6.76
	ymm		6.11	6.35	6.80
vmovdqa32	zmm		6.85	6.96	7.45
vmovdqu	xmm		6.10	6.27	6.47
	ymm		6.21	6.44	6.81
vmovdqu32	zmm		6.85	6.98	7.77
vmovdqa	xmm	64-bit integer	6.11	6.30	6.75
	ymm		6.20	6.36	6.74
vmovdqa32	zmm		6.85	7.07	7.80
vmovdqu	xmm		6.09	6.37	6.79
	ymm		6.18	6.40	6.80
vmovdqu32	zmm		6.85	7.09	7.80

Figure 4 shows a relationship plot of some of the data in Table 2. Missing are error bounds on the mean execution time values. As in Figure 2 a coloured line is used to group register type and *data alignment* together.

The high execution time shown in Table 2 for 32-bit floating point data in `zmm` registers using aligned and unaligned transfer instructions is interesting. In both alignment instruction cases, the variation about the mean value is small. However, such high execution times are substantially less than the corresponding memory/register execution times in Table 1.

The data in Table 2 and Figure 4 suggests:

- there is little difference in register to register transfer rate when using aligned or unaligned instructions
- transfer in `xmm` and `ymm` registers is faster than for `zmm`
- for integer data transfer, the total size of the data transferred appearred to influence the transfer time
- 64-bit floating point and 16-bit integer transfer speeds are approximately equal

Although the transfer times for `xmm` and `ymm` registers are faster than for `zmm` registers, they are moving far less data.

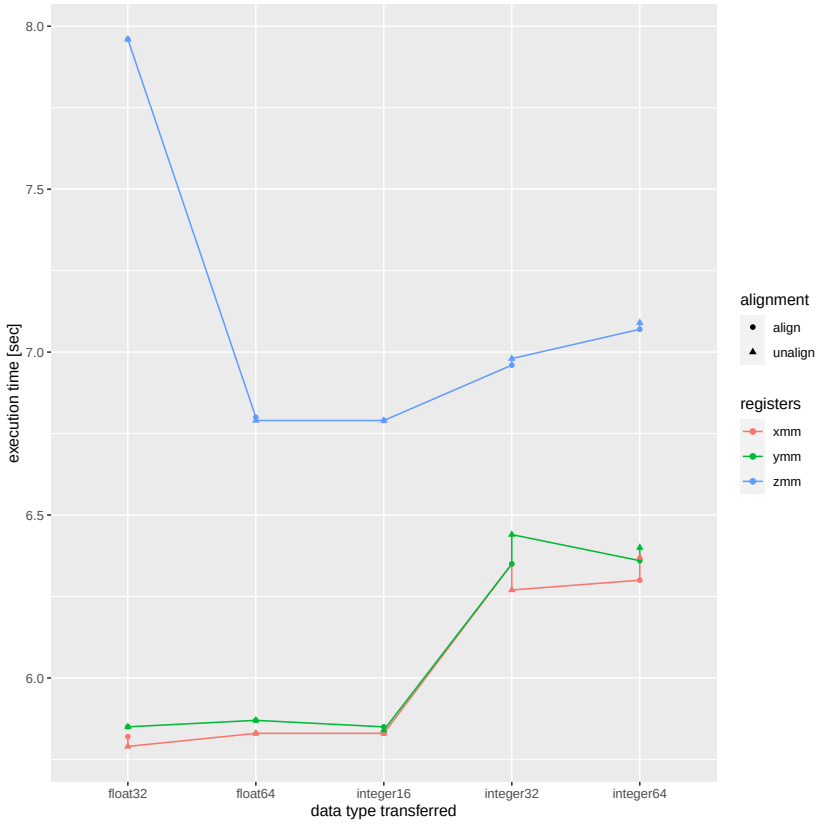

Figure 4: Relationships between SIMD register to register transfer data

1.3 Scalar processing for comparison

If vector processing using SIMD was not done, then individual data values would be the alternative. Such single values, like with SIMD, need to be moved into, out of, and between registers. A comparison measure of time to move single values is examined here.

As opposed to SIMD, processing of individual (scalar) value integer and floating point values is handled by different types of registers; integers use one type of register and floating point another type. Integers are handled using the CPU's `rxx` registers while floating point use the SIMD `xmm` registers. In using the `xmm` registers, floating point is handled in a different manner than in SIMD processing. In scalar processing a single floating point value is held in each `xmm` register as opposed to the SIMD case where a number of values are in the one register. This behaviour is the same as handling of integers in a CPU's 64-bit `rxx` registers.

Figure 5 shows an example of the C mainline and assembler function used to measure the execution time of processing scalar data. This particular example is set up to measure 32-bit integer data movement between registers where the data could be processed. Commented out in the two assembler functions is the code used to load floating point data to perform the same operation as the integer operation. The reason for the simplicity (one instruction as opposed to two) is C passes floating point values through the `xmm` registers.

```
#include  <stdio.h>
#include  <time.h>

extern float simd_io(int, int);
extern void between(void);

int  main()
{
  long  i;
  double  tick;
  double  my_get_wtime(void);
  int x[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
  int z[] = {11, 12, 13, 14, 15, 16, 17, 18, 19, 101};

  simd_io(x[2], z[5]);
  tick = my_get_wtime();     *
  for (i = 0; i < 5000000000; i++)  between();     **
  tick = my_get_wtime() - tick;
  printf("tick = %lf\n", tick);
  return(0);
}

double my_get_wtime()
{
  struct timespec  ts;
  double           time, nano;

  clock_gettime(CLOCK_MONOTONIC, &ts);
  time = (double) ts.tv_sec;
  nano = (double) ts.tv_nsec;
  return (time + nano/1000000000.0);
}

; moving data in and out of registers

        global  simd_io
        global  between

        section .text

simd_io:
       mov    rbx, rdi      ; put first integer argument into rax
       mov    rcx, rsi      ; put second integer argument into rcx
;       movss    xmm0, xmm1
       ret

between:
       mov    rax, rbx        ; copy data in rbx to rax
       mov    rcx, rax        ; copy data in rax to rcx
;       movss  xmm8, xmm0      ; copy xmm0 data to xmm8
;       movss  xmm5, xmm0      ; copy xmm0 data to xmm8
       ret
```

Figure 5: Composite C and assembler code to measure scalar register speed

The statements flagged in Figure 5 with * and ** were changed when measuring moving data in and out of registers and between registers. When measuring data moving in and out of registers the * statement was removed. For those measurements the ** statement was replaced by:
`for (i = 0; i < 5000000000; i++) simd_io(x[2], z[1]);`.
The `x[2]` and `z[10]` represent passing of discrete values. With integer operations, the type and values stored in the `x[]` and `z[]` arrays were changed appropriately.

In contrast to the SIMD codes of Figure 1 and Figure 3, scalar processing as in Figure 5 pass individual values to the assembler function.

As in the SIMD experiments, each the assembler function performed two similar processing operations. These experiments were performed using the same computing environment of MacPro computer, `clang` compiler, and `nasm` assembler.

Table 3: Timing results for moving SIMD data types as scalar values

register operation	instruction used	data type	Execution time [sec] min	mean	max
in/out	movss	32-bit float	6.87	6.89	6.90
		64-bit float	6.89	6.93	6.98
	mov	16-bit integer	7.01	7.07	7.15
		32-bit integer	6.93	6.95	7.03
		64-bit integer	6.92	6.95	7.07
between	movss	32-bit float	5.74	5.77	5.83
		64-bit float	6.00	6.19	6.91
	mov	16-bit integer	6.09	6.14	6.27
		32-bit integer	6.10	6.19	6.26
		64-bit integer	5.85	6.38	6.87

The execution times measured are contained in Table 3. Since they were obtained in a manner similar to those of the above SIMD movement experiments, the results of all experiments should be comparable.

The results in Table 3 indicate movement of integer and floating point data is very similar despite integers using `rxx` CPU registers and floating point using `xmm` registers. Such movement is slower between memory and registers than between members of the same register class.

2 Summary

Table 1 and Table 2 are indicative of relative speed of SIMD execution. Each call of the function containing the SIMD instruction of interest was executed 5,000,000,000 times. The function itself contained two similar instructions. The timing also covered entry to, and return from the function call. The method of timing, the manner of execution of the function call by a do-loops, the number of repeated calls of the function, and the composition of the function were also the same. So the execution times in those tables should be indicative of the relative performance of the SIMD instructions measured.

The execution time results in Table 1 suggests there is only marginal advantage of using aligned register/memory data transfer. The occurrence of faults resulting from aligned data transfers supports use of unaligned transfers.

Table 4 contains mean execution times for vector and scalar movement of data for the six types

supported by SIMD operations extracted from Tables 1, 2 and 3. The vector data is for unaligned data transfers. The tabled values indicate the vector operation is slower than the corresponding scalar operation. However, the scalar operation needs to be performed multiple times to move the same number of data values as a single vector operation. Once in the registers, one vector operation processes all those values in contrast to multiple operations on scalar values. This data suggests there is a speed advantage to be achieved by using SIMD vector processing and minimizing the transfer of all data types between all SIMD register types and memory, i.e. the CPU assembly philosophy of retaining data in registers also applies to SIMD registers.

Table 4: Summary of execution times

	memory-register				register-register			
	vector			scalar	vector			scalar
data type	xmm	ymm	zmm		xmm	ymm	zmm	
32-bit float	7.23	7.02	8.44	6.89	5.79	5.85	7.96	5.77
64-bit float	6.97	6.95	8.04	6.93	5.83	5.87	6.79	6.19
16-bit integer	5.85	5.94	8.49	7.07	5.83	5.84	6.79	6.14
32-bit integer	7.23	7.24	8.04	6.95	6.27	6.44	6.98	6.19
64-bit integer	6.98	6.96	8.05	6.95	6.37	6.40	7.09	6.38

This work supports the basics learnt using assembler with a CPU as also being applicable to SIMD processing. Data is processed in registers. The number of movements of data into and out of registers should be minimized. To do this maximizing the number of registers used. Where possible data and the results of processed data should be retained in registers if such data are to be reused in other operations.

References

AMD (2020), "AMD64 Architecture Programmer's Manual Volume 4: 128-Bit and 256-Bit Media Instruction", version 3.24, AMD64 Technology, `https://www.amd.com/system/files/TechDocs/26568.pdf`, accessed Oct. 23, 2021.

ORACLE (2020), "x86 Assembly Language Reference Manual", `https://docs.oracle.com/cd/E26502_01/pdf/E28388.pdf`, accessed Sept. 28, 2021.