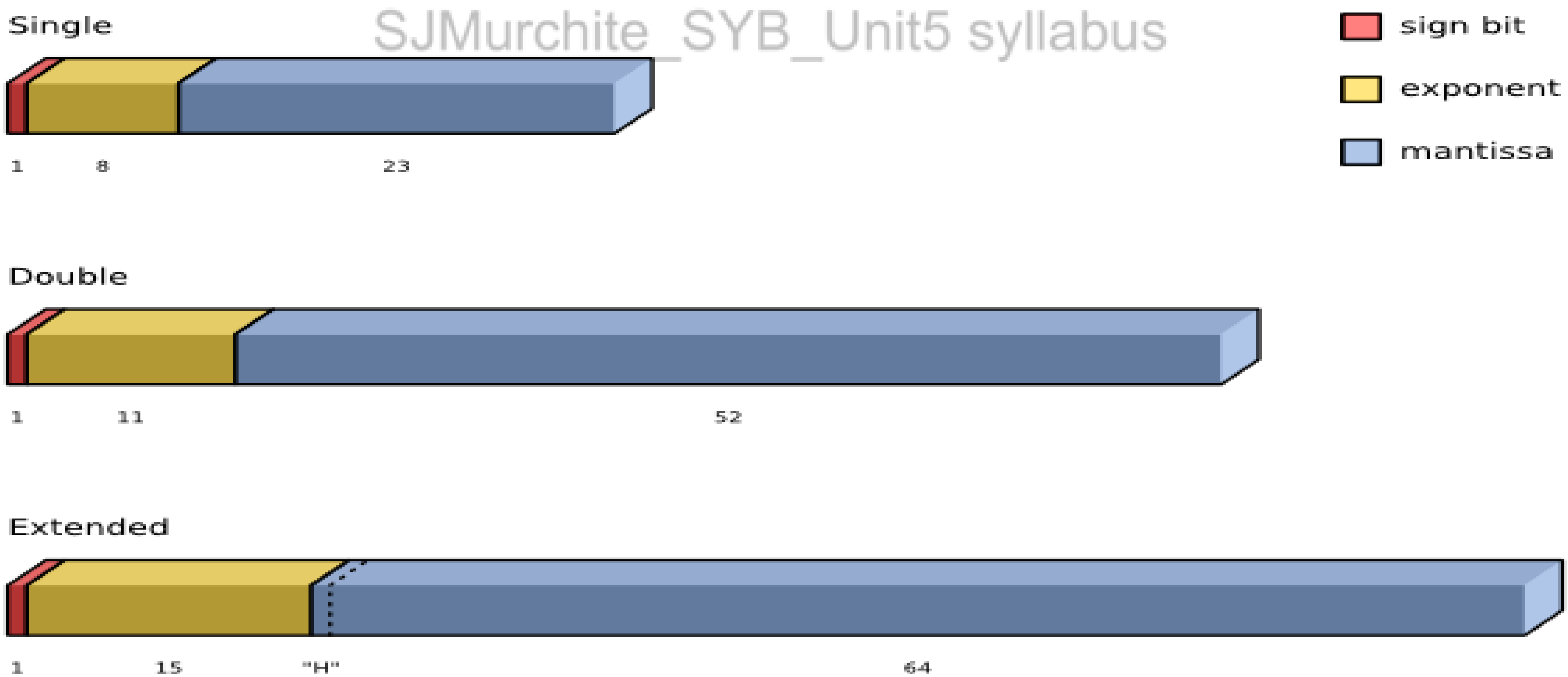
Unit5

SJMurchite_SYB_Unit5 syllabus

Floating point and SIMD instructions

Floating point Representation

- There are three kinds of floating point numbers in IA32 architectures.
- IEEE754 Single precision floating point format
- IEEE754 Double precision floating point format
- Double extended precision floating point number



Floating point Representation

- IA32 architectures support IEEE754 number representation schemes of floating point numbers. There are three kinds of floating point numbers in IA32 architecture.
- • IEEE754 Single precision floating point format
- • These are 32 bit wide numbers and have three parts-a sign, a mantissa and an exponent.
- • Sign is 1 bit wide and represents the sign of the number. Mantissa is 23 bit wide and represents the fractional part of the floating point number in binary. A leading one bit integer is not stored and is assumed as 1 in the normalized numbers.
- • Exponents are stored in excess-127 representation.
- • IEEE754 Double precision floating point format
- • These are 64 bit wide number with sign ,mantissa and exponent in excess 1023 representation.
- • Double extended precision floating point number
- • Double extended precision floating point numbers are 80 bit wide.
- • This number format is used to perform computations internally in IA32 processors when instructions in x87 floating point instruction set are used.

Floating point conversion in single precision

- Conversion of decimal number to IEEE single precision floating point format
- -24.75

Sign bit(S) =1   (for negative number)

$24.75 = 11000.11 * 2^0$

$1.100011 * 2^4$

- Exponent is excess-127 in single precision format

bias exponent = actual exponent + bias value

=127+4=131

Mantissa (M)=(1.100011)

1	10000011	10001100000000000000000

-24.75 = 0xC1C60000

+7.5=   ?

Floating point conversion in double precision

- Conversion of decimal number to IEEE single precision floating point format
- -24.75

Sign bit(S) =1 (for negative number)

$24.75 = 11000.11 * 2^0$

$1.100011 * 2^4$

- Exponent is excess-1023 in double precision format

bias exponent = actual exponent + bias value

=1023+4=1027

Mantissa (M)=(1.100011)

1	10000000011	100011000000000000000000000000000000000000000000000

-24.75 = 0xC038 C000 0000 0000

+7.5= ?

First Floating point program

```
.section .data
 value1:
  .float   -24.75
  value2:
  .double   -24.75
  .section .bss
  .lcomm  data ,4
  .lcomm   data1, 8
.section .text
.globl  _start
  _start : nop
finit
flds value1
  fldl value2
  fsts data
  fstl data1
  movl $1,%eax
  movl $0,%ebx
  int $0x80
```

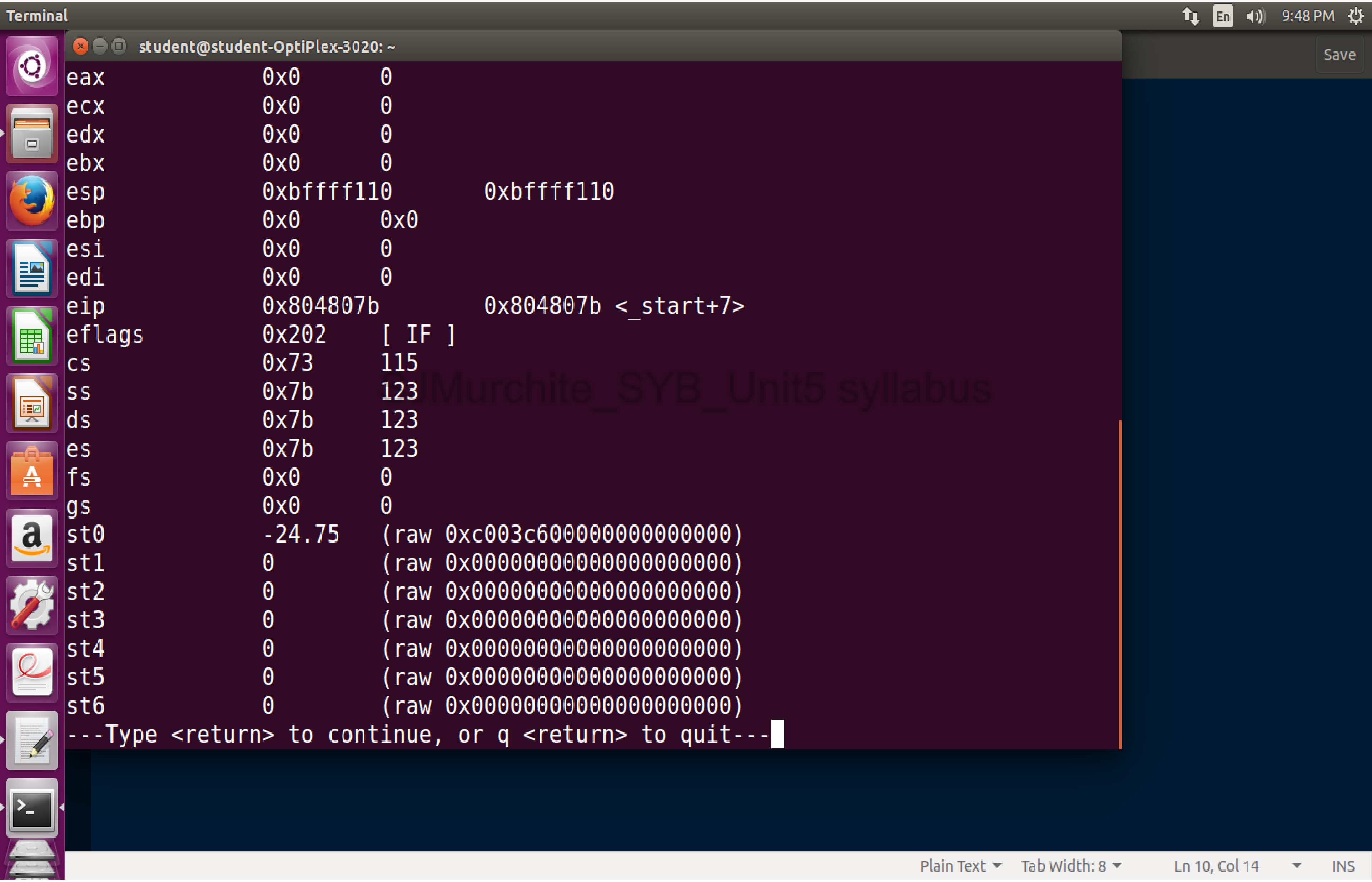
Single and double precision Number in IEEE

Terminal

student@student-OptiPlex-3020: ~

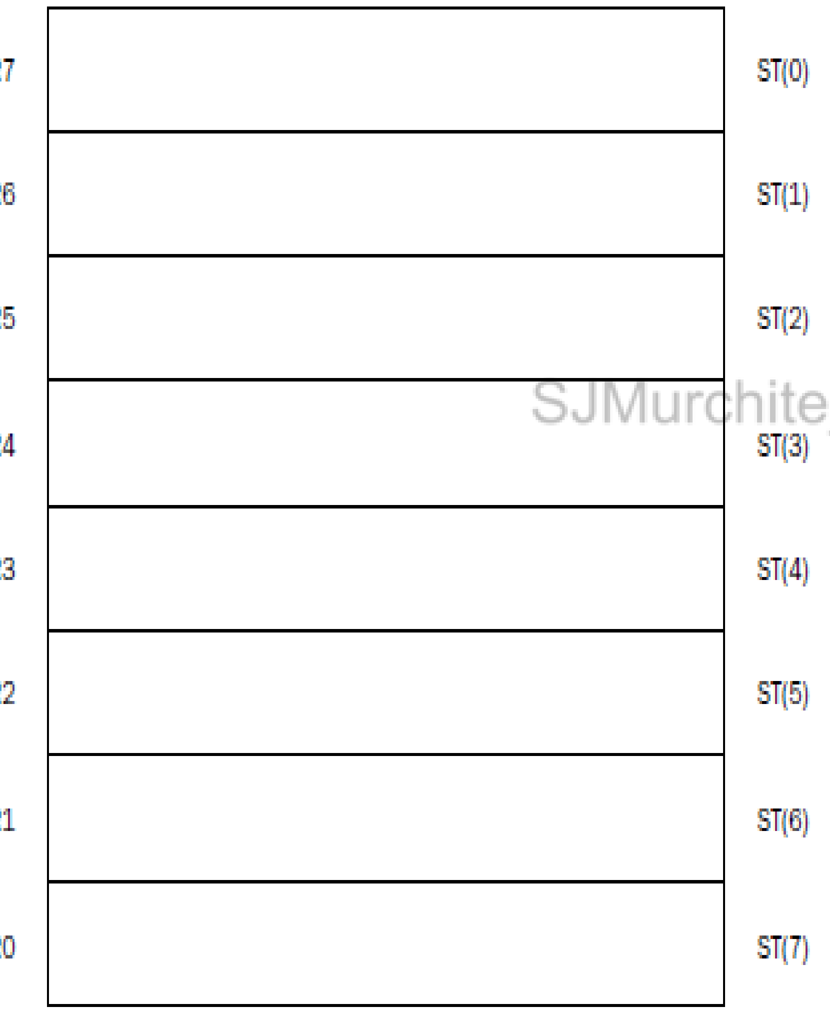
```
student@student-OptiPlex-3020:~$ as -gstabs -o float1.o float1.s
student@student-OptiPlex-3020:~$ ld -o float1 float1.o
student@student-OptiPlex-3020:~$ gdb -q float1
Reading symbols from float1...done.
(gdb) break*_start+1
Breakpoint 1 at 0x8048075: file float1.s, line 11.
(gdb) run
Starting program: /home/student/float1

Breakpoint 1, _start () at float1.s:11
11          flds value1
(gdb) x/fx &value1
0x8049093:      0xc1c60000
(gdb) s
12          fldl value2
(gdb) x/gfx &value2
0x8049097:      0xc038c00000000000
(gdb) s
13          fstl data
(gdb) s
14          movl $1,%eax
(gdb) x/gfx &data
0x80490a0 <data>:       0xc038c00000000000
(gdb)
```

Double extended precision Number in IEEE



16 april2021 9:40am

Architecture of floating point processor



- **X87 general purpose Data registers in IA32 processor**

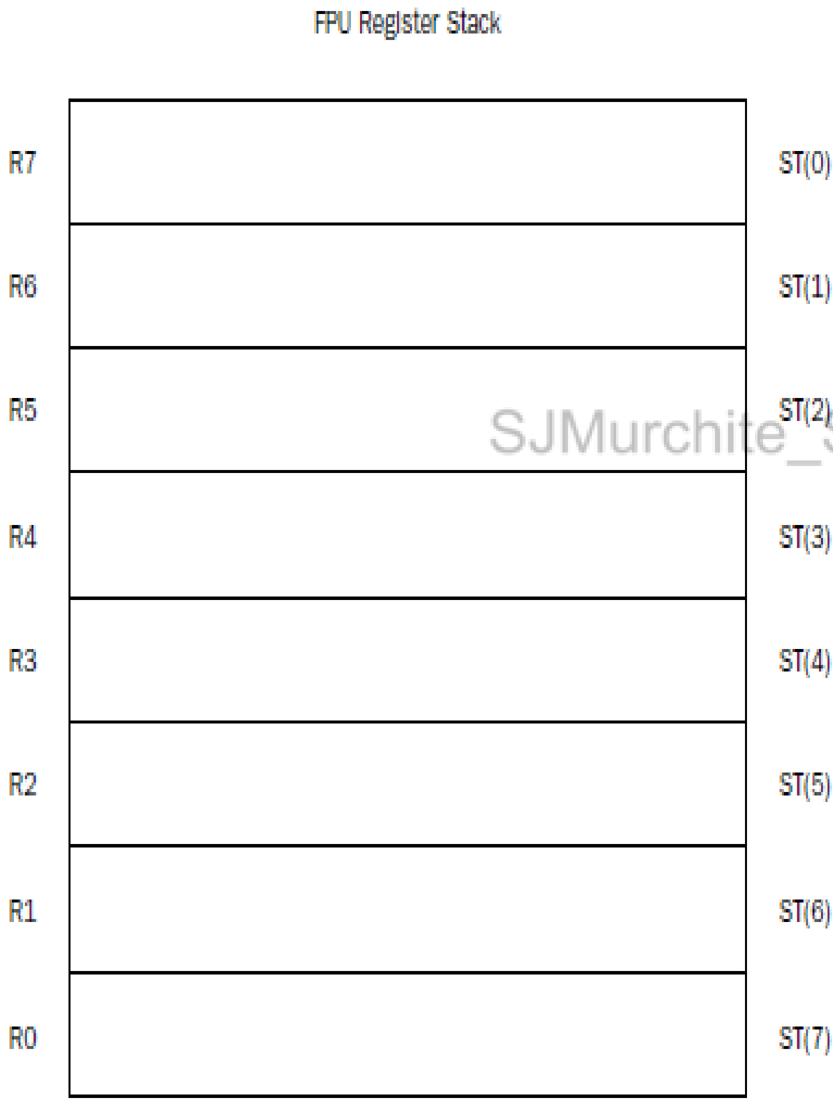
It include 8 general purpose data registers, each 80 bit wide and capable of storing a real number in double extended precision format.

- As data is loaded into the FPU stack, the stack top moves downward in the registers.

- When eight values have been loaded into the stack, all eight FPU data registers have been utilized.

- If a ninth value is loaded into the stack, the stack pointer wraps around to the first register and replaces the value in that register with the new value

Architecture of floating point processor



- When an integer ,BCD, single and double precision operand is loaded into one of these data registers, the operand is implicitly converted to double extended precision format.
- In addition to data registers, x87 also includes three 16 bit registers which are used in controlling the computations.

- These registers are the following
  - X87 control register
  - X87 status register
  - X87 tag register

Top of stack

Stack register	Data register when TOP=5	Data register when TOP=2
St(0)	R5	R2
St(1)	R4	R1
St(2)	R3	R0
St(3)	R2	R7
St(4)	R1	R6
St(5)	R0	R5
St(6)	R7	R4
St(7)	R6	R3

The register stack used registers in a circular buffer.
1) When TOP=5, then st (0) refers to x87 data register R5.

Floating point exceptions

- Invalid operation exception (IE)
  - It is raised when operands of the instruction contain invalid values such as NaN.
  - Example: division of 0 by 0 should lead to NaN
- Divide by zero exception (ZE)
  - Occurs when divisor contains a zero and dividend is a finite number other than 0
- De-normalized operand exception (DE)
  - It is raised when one of the operands of an instruction is in denormalized form.
  - De-normalized numbers can be used to represent numbers with magnitudes too small to normalize (i.e. below 1.0×2^{-126})
- Numeric overflow exception (OE)
  - Occurred when rounding result would not fit into destination operand.
  - *When a double precision floating point number is to be stored in single precision floating point format, a numeric overflow is set to have occurred.*
- Numeric underflow exception (UE)
  - Occur when the instruction generate result whose magnitude is less than the smallest possible normalized number.
- Precision (Inexact result) exception (PE)
  - Division of (1.0/12.0) results in recurring infinite sequence of bits.
  - This number cannot be stored in any format without loss of precision.

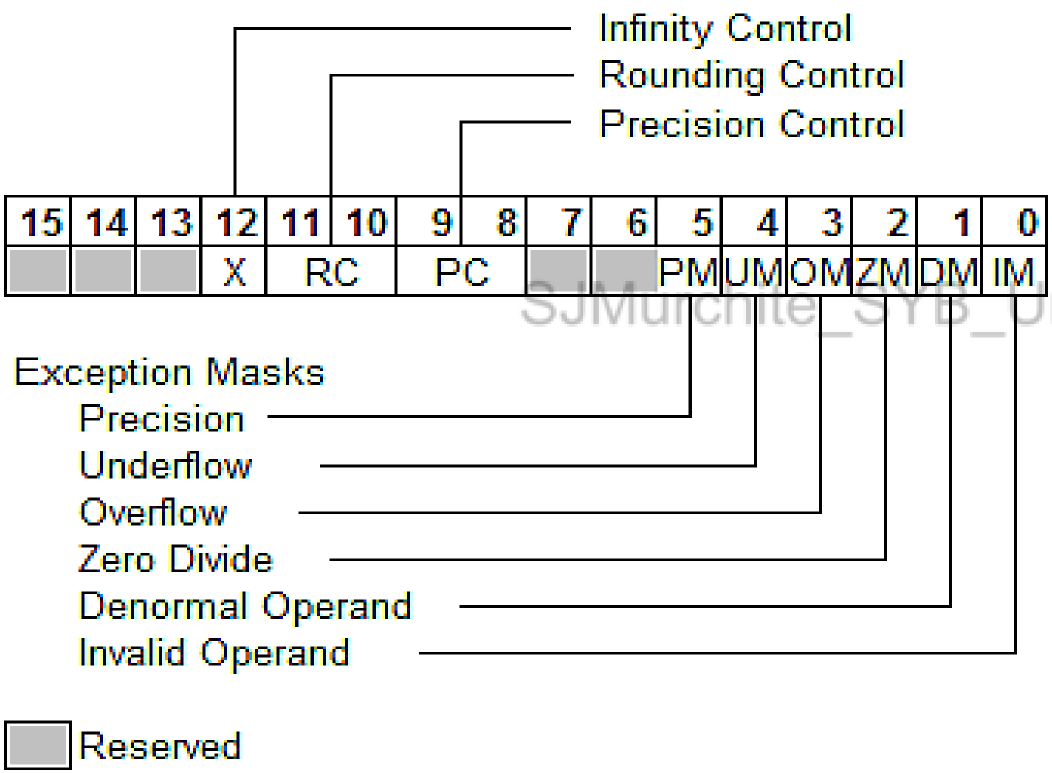
X87 control register

During computation ,several floating point error conditions such as divide by zero occur
In case of such errors , x87 FPU can be programmed to raise floating point exception

Bit 0 to bit 5
0 indicate : unmask exception
1 indicate: Mask excpetion

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			X	RC		PC				PM	UM	OM	ZM	DM	IM

X87 control register



Used to handle the precision and rounding of operands

Infinity bit always set to 0

Precision control

00- single precision

01 unused

10-Double precision

11-Double extended precision

Rounding control

00-Round to nearest

01 Round down

10-Round up

11 Round towards zero(Truncate)

Exception Mask

0-Unmask

1-Mask

Rounding Controls in IA 32 architectures

- +1.00010010001101001110 1 0011

1. Rounding towards 0 +1.000100100011010011101
2. Rounding towards + ∞ (Rounding up) +1.000100100011010011110

- Rounding towards plus infinity control used 0011 would be removed but a 1 will be added to the LSB of remanding bits because the removed bit pattern is other than a zero.

- +1.000100100011010011101 1011

3. Rounding towards int +1.000100100011010011110
4. Rounding towards - ∞ (Rounding down) +1.000100100011010011101

Note:1) Rounding towards 0 identical as rounding towards -∞

Note:1) Rounding towards int identical as rounding towards +∞

X87 Status register

1) TOP-it is 3 bit field contain index to register assumed to be at the top of the stack.

2) Four conditional code flags **C0 to C3**

A floating point number can be compared in different ways with other floating point number.

The result of comparison is stored in either eflags or in floating point conditional code c0-c3

3) SF-The stack fault bit in status register indicates errors due to stack overflow or underflow conditions

Example: when one data on stack and addition operation is performed which takes two data from the stack, a stack fault occurs.

Stack overflow; when all 8 registers are in use and an attempt is made to load an data , stack overflow occurs.

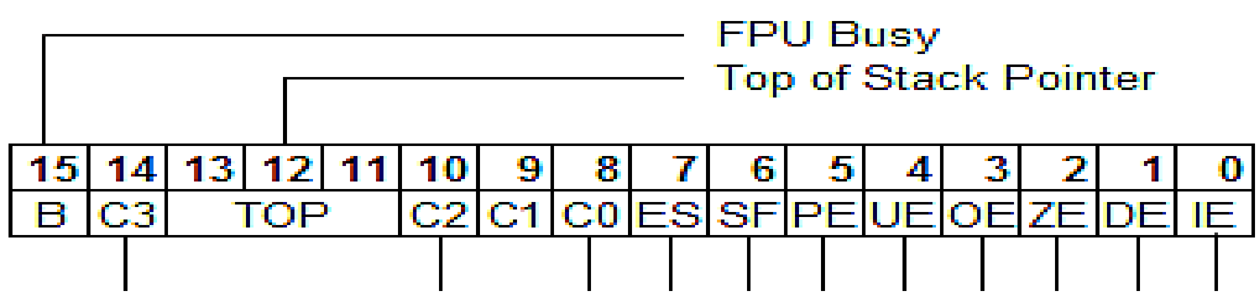
ES- Error status

1- Error occurred   0- No error

B bit

FPU Busy bit B=0 FPU free

FPU Busy bit B=1 FPU busy.



X87 Tag register

16-bit Tag Register

15							0
R7	R6	R5	R4	R3	R2	R1	R0

Each data register of x87 FPU has an associated 2 bit tag contained in a tag register.
The code in the tag register indicates the type of the value of the corresponding data register.

Tag is associated with data register
00- R has a valid value
01-R has a zero
10-R has a special value (Nan,+/- infinity, renormalized number)
11-R is empty.

Program to Illustrate Status Register, Tag register and Control register.

```
.section .bss


.lcomm status,2

.section .text

.globl _start
_start : nop

        fstsw status

   movl $1,%eax
   movl $0,%ebx
   int $0x80
```

```
student@student-OptiPlex-3020: ~
    0x00000000000000000000000000000000, 0x00000
ymm2           {v8_float = {0x0, 0x0, 0x0, 0x0,
  v4_double = {0x0, 0x0, 0x0, 0x0}, v32_int8 =
  v16_int16 = {0x0 <repeats 16 times>}, v8_int3
---Type <return> to continue, or q <return> to
[1]+  Stopped                 gdb -q fs
student@student-OptiPlex-3020:~$ gdb -q fs
Reading symbols from fs...done.
(gdb) break*_start+1
Breakpoint 1 at 0x8048075: file fs.s, line 11.
(gdb) run
Starting program: /home/student/fs

Breakpoint 1, _start () at fs.s:11
11              fstsw status
(gdb) s
13          movl $1,%eax
(gdb) print/x $ftag
$1 = 0xffff
(gdb) print/x $fctrl
$2 = 0x37f
(gdb) print/x $fstat
$3 = 0x0
(gdb)
```

20 april,2021(Tuesday)

Basic arithmetic instructions

Fadd

Add two floating point number available in stack registers

Example   fadd %st(1),%st*0)

Here %st(0) is added with %st(1) and result stored in destination stack %st(0)

Fiadd

It is used to add add an integer stored in memory location to the floating point number on the top of stack.

Faddr

Add two floating point number available in stack registers

Example faddr %st(1),%st(0)

Here %st(0) is added with %st(1) and result stored in source stack register %st(0)

Faddp

Add two floating point number available in stack registers and pop the top of stack.

Example faddp %st(1),%st(0)

here %st(0) is added with %st(1) and remove stack registers and put result in top of stack.

$$((43.65 / 22) + (76.34 * 3.1)) / ((12.43 * 6) - (140.2 / 94.21))$$

```
.section .data
value1:
   .float 43.65
value2:
   .int 22
value3:
   .float 76.34
value4:
   .float 3.1
value5:
   .float 12.43
value6:
   .int 6
value7:
   .float 140.2
   value8:
    .float 94.21
.section .text
```

```
.globl _start
_start:
nop
   finit
   flds value1
   fidiv value2
   flds value3
   flds value4
   fmul %st(1), %st(0)
   fadd %st(2), %st(0)
   flds value5
   fimul value6
   flds value7
   flds value8
   fdivrp
   fsubr %st(1), %st(0)
   fdivr %st(2), %st(0)
  movl $1,%eax
  movl $0,%ebx
  int $0x80
```


1	2	3	4	5	6
43.65	1.98409	76.34	3.1	236.654	238.63809
		1.98409	76.34	76.34	
			1.98409	1.98409	

7	8	9	10	11	12	13
12.43	74.58	140.2	94.21	1.48816	73.09184	3.264907
238.63809	238.63809	74.58	140.2	74.58	74.58	
		238.63809	74.58	238.63809	238.63809	
			238.63809			

$$((43.65 / 22) + (76.34 * 3.1)) / ((12.43 * 6) - (140.2 / 94.21))$$

1	2	3	4	5	6
43.65	1.98409	76.34	3.1	236.654	238.63809
		1.98409	76.34	76.34	
			1.98409	1.98409	

7	8	9	10	11	12	13
12.43	74.58	140.2	94.21	1.48816	73.09184	3.264907
238.63809	238.63809	74.58	140.2	74.58	74.58	
		238.63809	74.58	238.63809	238.63809	
			238.63809			

Advanced Floating-Point Maths

- .section .data
- value1:
-   .float 395.21
- value2:
-   .float -9145.290
- value3:
-   .float 64.0
- .section .text
- .globl _start
- _start:
-   nop
- finit
-   flds value1
-   fchs
-   flds value2
-   fabs
-   flds value3
-   fsqrt
-   movl $1, %eax
-   movl $0, %ebx
-   int $0x80

Fchs this instruction is used to change the sign of the input argument on the top of stack i.e st(0)
Fabs this instruction is used to compute the absolute value of register st(0)
Fsqrt this instruction is used to compute square root of a number on top of stack st(0) and returns the result in register st(0) overwriting the value stored previously

Roots of a quadratic equation

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

$$x^2-4x+3$$

```
.section .text

.globl _start
_start : nop

        flds a
        fimul val1
        flds c
        fmul %st(1),%st(0)
        flds b
        fmul %st(0),%st(0)
        fsubp %st(1),%st(0)
        fsqrt
        flds b
        fchs
        fadd %st(1),%st(0)
        flds b
        fchs
        fsub %st(2),%st(0)
        flds a
        fimul val2
        fsts val5
        fdivr %st(2),%st(0)
        fsts val3
        flds val5
        fdivr %st(2),%st(0)
        fsts val4

   movl $1,%eax
   movl $0,%ebx
   int $0x80
```

```
(gdb) s
38                  flds a
(gdb) s
39                  fimul val2
(gdb) s
40                  fsts val5
(gdb) s
41                  fdivr %st(2),%st(0)
(gdb) s
42                  fsts val3
(gdb) s
43                  flds val5
(gdb) x/f &val3
0x80490f8 <val3>:       3
(gdb) s
44                  fdivr %st(2),%st(0)
(gdb) s
45                  fsts val4
(gdb) s
47              movl $1,%eax
(gdb) x/f &val4
0x80490fc <val4>:       1
(gdb)
```

26 April ,2021 11: 50 am

constant loading x87 FPU instructions

in the x87 instruction set, there are certain instructions that can load commonly used constants on the stack top. These instructions are the following

Fld1	st(0)=1.0
Fldz	st(0)=0.0
Fldpi	st(0)=3.14
Fldl2e	$\log_2 e$
Fldln2	$\log_e 2$
Fldl2t	$\log_2 10$
Fldlg2	$\log_{10} 2$

Trigonometric x87 FPU instructions

In the x87 FPU instruction set, following instructions are available and can compute trigonometric functions for arguments stored on the stack top

.

- Fsin It compute sin of angle in radians provided in st(0)

-
- Fcos It compute cosine of angle in radians provided in st(0)

- Fsincos It compute and leaves two values in the register stack.
-
- Fptan It removes the angle provided on top of the register stack and computers its tangent.

radians = (degrees * pi) / 180

- .section .data
- degree1:
- .float 90.0
- val180:
- .int 180
- .section .bss
- .lcomm radian1, 4
- .lcomm result1, 4
- .lcomm result2, 4
- .section .text
- .globl _start
- _start:
- nop
- finit
- flds degree1
- fidivs val180
- fldpi
- fmul %st(1), %st(0)
- fsts radian1
- fsin
- fsts result1
- flds radian1
- fcos
- fsts result2
- movl $1, %eax
- movl $0, %ebx
- int $0x80

logarithmic x87 FPU instructions

there are two instructions in x87 FPU instruction set to compute logarithm

Fyl2x
it compute ylog2x with y and x being in register st(1) and st(0) respectively.

fyl2xpl
it compute ylog2 (x+ 1) with y and x being in register st(1) and st(0) respectively

Both the instructions take two implied arguments x and y on the register stack in registers st(0) and st(1) respectively.
After successful execution of this instruction ,both operands are removed and result is pushed on the register stack.

$$\log_b X = (1/\log_2 b) * \log_2 X$$

```
.section .data
value:
   .float 12.0
base:
   .float 10.0
.section .bss
   .lcomm result, 4
.section .text
.globl _start
_start:
   nop
 finit
   fld1
   flds base
   fyl2x
   fld1
   fdivp
   flds value
   fyl2x
   fsts result
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

SIMD Technology

- **Introduction**
- Intel introduced multimedia extension (MMX) instruction set with Pentium processors.
- Instruction operate simultaneously on multiple data values.
- Application include image processing, voice and data communication.
- In Pentium III, intel introduced streaming SIMD extension (SSE) instruction set.
- While MMX instructions operate on integer data, SSE instructions operate on floating point data.
- The SSE instruction set is targeted at applications that operate on large arrays of floating point numbers such as 3D graphics, video encoding and decoding etc.

SIMD Environment

- SIMD technology provides additional way to define integers.
- It Perform arithmetic operations on a group of multiple integers simultaneously
- SIMD architecture uses packed data type
- A packed integer is series of bytes that can represent more than one integer.
- SIMD instruction set provide a few register called MMX registers and XMM registers.
- MMX registers includes 8,64 bit registers named mm0-mm7.

- Data register are 80 bit wide, out of which only 64 bits are used by instructions in the SIMD instruction set.
- It is recommended to use finit instruction to initialize x87 FPU at the time of switching from SIMD to x87 environment.
- XMM registers includes 8,128 bit registers named xmm0-xmm7.
- XMM registers support integer and floating point data types.
- Supported floating point data type include 4 packed single precision floating point numbers.

Loading and retrieving packed integer values

```
.section .data
packedvalue1:
.byte 10, 20, -30, 40, 50, 60, -70, 80
packedvalue2:
.short 10, 20, 30, 40
packedvalue3:
.int 10, 20
.section .text
.globl _start
_start:
movq packedvalue1, %mm0
movq packedvalue2, %mm1
movq packedvalue3, %mm2
```

MMX addition and subtraction instructions

- With normal addition and subtraction with general-purpose registers, if an overflow condition exists from the operation, the EFLAGS register is set to indicate the overflow condition

- when using MMX addition or subtraction, you must decide ahead of time what the processor should do in case of overflow conditions within the operation.

- You can choose from three overflow methods for performing the mathematical operations:

1. Wraparound arithmetic
2. Signed saturation arithmetic
3. Unsigned saturation arithmetic

Examples of SIMD byte operations with saturations

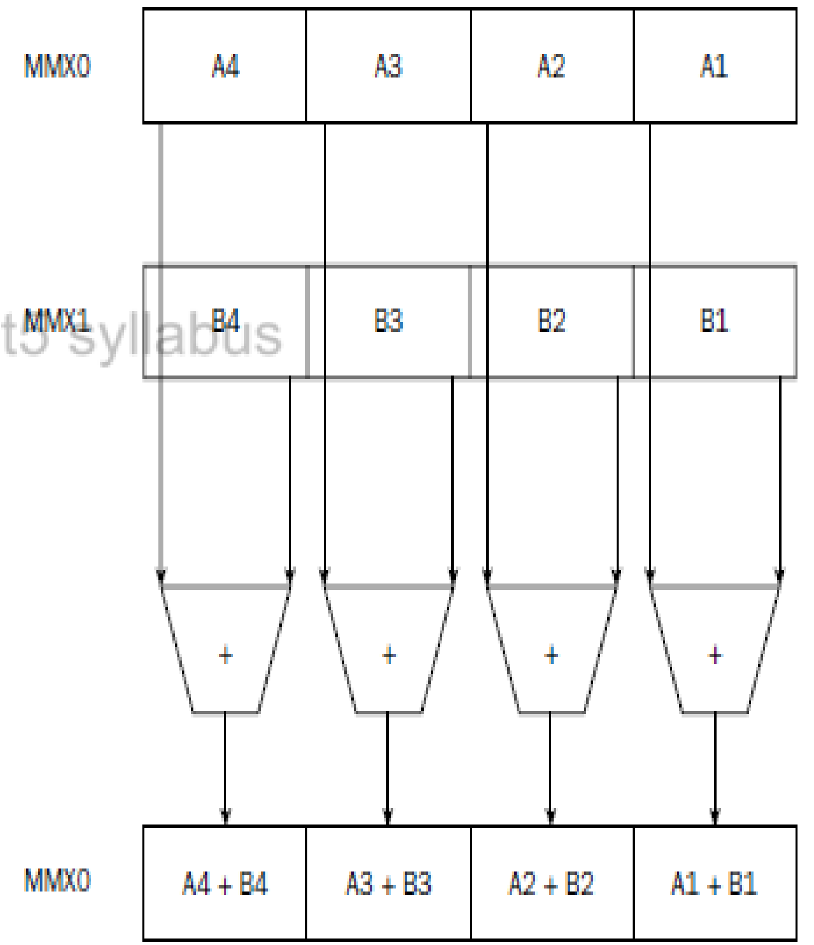
A	B	Operation	Wrap around	Signed saturation	Unsigned saturation
0xA3	0xC2	A+B	0x65	0x80	0xFF
0x57	0xB2	A+B	0x09	0x09	0xFF
0x78	0x40	A+B	0xB8	0x7F	0xB8
0x40	0x72	A-B	0xCE	0xCE	0x00

Wrap around- Whenever the result is larger than 0xFF, Extra bit is dropped. Result is treated as module 256.
Signed saturation- when -93 and -62 in decimal added. The summation of two numbers will be -155 which is below the minimum possible value 0x80
Unsigned saturation- When 163 and 194 in decimal added the result is 357, which again cannot represent in 8 bit unsigned number format. So the result is 0xFf

MMX addition

```
.section .data
value1:
.int 10, 20
value2:
.int 30, 40
.section .bss
.lcomm result, 8
.section .text
.globl _start
_start: nop
```

```
movq value1, %mm0
movq value2, %mm1
paddd %mm1, %mm0
movq %mm0, result
movl $1, %eax
movl $0, %ebx
int $0x80
```

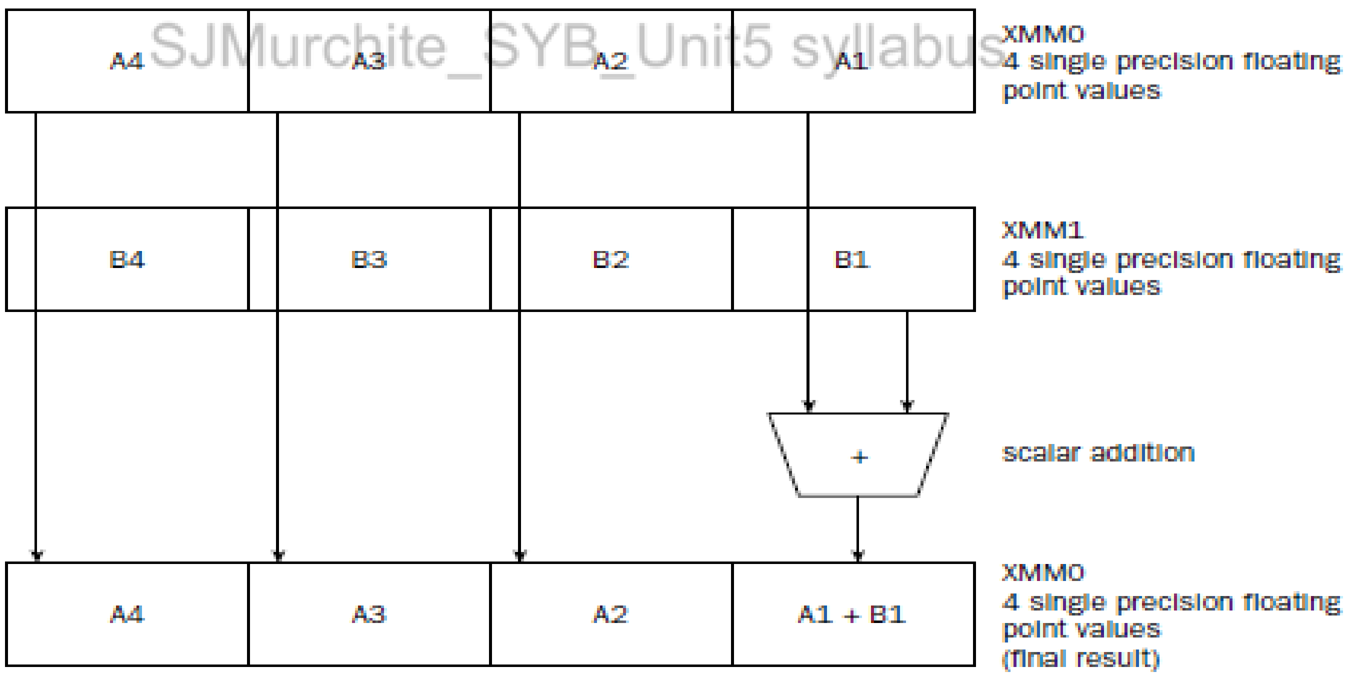



PADDD Add packed double-word integers with wraparound
PADDSD……………….with sign saturation
PADDUSD………………with unsigned saturation

13 March 2020 3:30PM

SSE Instructions

- Main purpose of the SSE technology is to perform SIMD operations on floating point data.



DATA TRANSFER IN SSE TECHNOLOGY

- .section .data
- .align 16
- value1:
- .float 12.34, 2345.543, -3493.2, 0.4491
- .section .text
- .globl _start
- _start:
- movaps value1, %xmm0

.align directive instructs the gas assembler to align the data on a specific memory boundary.
It takes a single operand, the size of the memory boundary on which to align the data

Addition Example of using SSE arithmetic instructions

- .section .data
- .align 16
- value1:
- .float 12.34, 2345., -93.2, 10.44
- value2:
- .float 39.234, 21.4, 100.94, 10.56
- .section .bss
- .lcomm result, 16
- .section .text
- .globl _start
- _start:
- nop
- movaps value1, %xmm0
- movaps value2, %xmm1
- addps %xmm1, %xmm0
- movaps %xmm0, result
- movl $1, %eax
- movl $0, %ebx
- int $0x80

Thank you