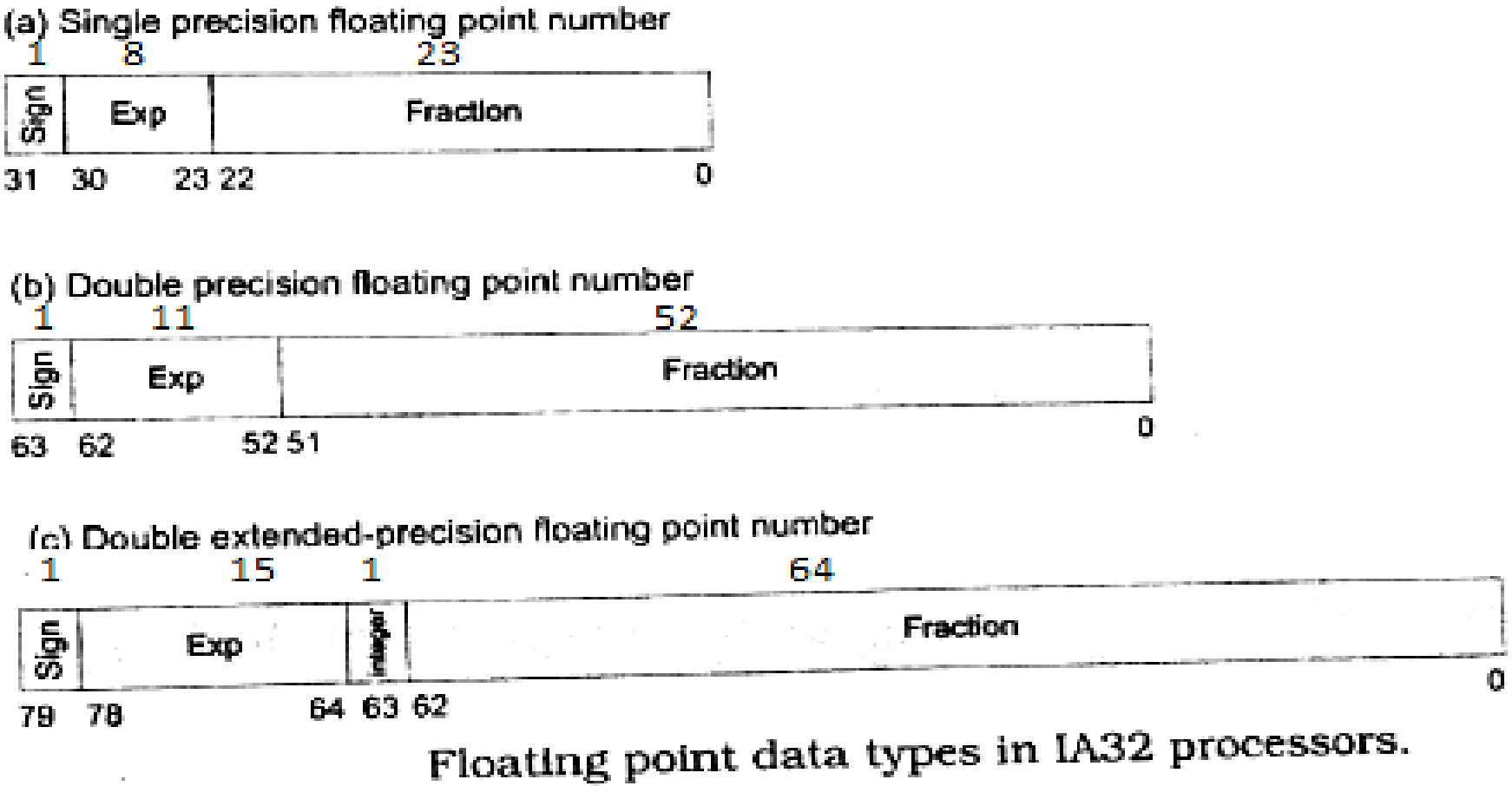
Unit 5

Floating point and SIMD instructions

Prepared By- Mrs . J. D. Pakhare

Floating point numbers in IA32 Architecture:

- IEEE754 Single precision floating point format
- IEEE754 Double precision floating point format
- Double extended precision floating point number



Floating point data types in IA32 processors.

Floating point conversion in single & double precision

- **single precision:**
- -24.75 --- convert to binary and normalize

$$= -1.100011 * 2^4$$

Biased exponent = 127+4 =131= $(10000011)_2$

- **Double precision:**
- -24.75 --- convert to binary and normalize

$$= -1.100011 * 24$$

- Biased exponent =1023+4 =1027 = $(10000000011)_2$

- Represent in single and double precision floating point format--

Floating point conversion in single & double precision

IEEE single precision	**Sign**	**Exp**	**Mantissa**
	1	10000011	1000110……..00
IEEE754 no in Hex	0xC1C60000		
IEEE double precision	**Sign**	**Exp**	**Mantissa**
	1	10000000011	10001100……0000
IEEE754 no in Hex	0xC038 C000 0000 0000		

- 0.0625--- convert to binary and normalize

$$= 0.0001*2^{0} = 1.0x2^{-4}$$

- Biased exponent = 127-4 =123 = $(01111011)_2$

IEEE single precision	Sign	Exp	Mantissa
	0	01111011	00000……..00
IEEE754 no in Hex	0x3D80 0000		
IEEE double precision	**Sign**	**Exp**	**Mantissa**
	0	01111111011	00000000……0000
IEEE754 no in Hex	0x3FB0 0000 0000 0000		

- Convert +7.5 into single and double precision FP format?

Moving floating-point values/data transfer

- FLD ----instruction is used to move floating-point values into and out of the FPU registers.

  fld source

  – source can be a 32, 64, or 80-bit memory location.

  flds---for single precision

  fldl---- for double precision

- FST ---instruction is used for retrieving the top value on the FPU register stack and placing the value in a memory location.

  fst dest

  fsts---for single precision

  fstl---- for double precision

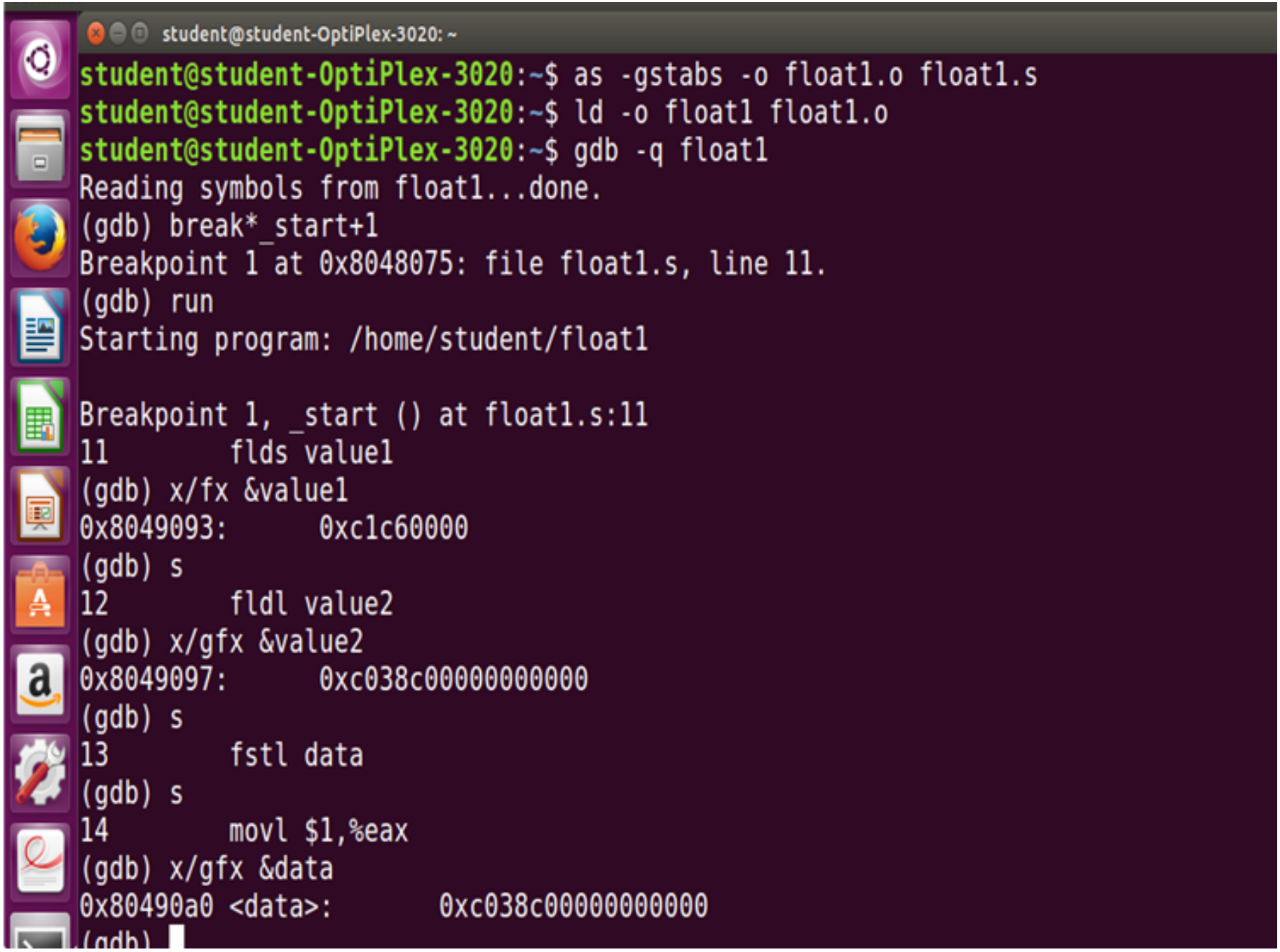
- fstp memvar
- fstp src

- Fild memvr
- Fist memvar
- Fistp memvar

Defining floating-point values--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
         flds value1
        fldl value2
        fsts data
         fstl data1
         movl $1,%eax
        movl $0,%ebx
        int $0x80
```


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)

Architecture of floating point processor

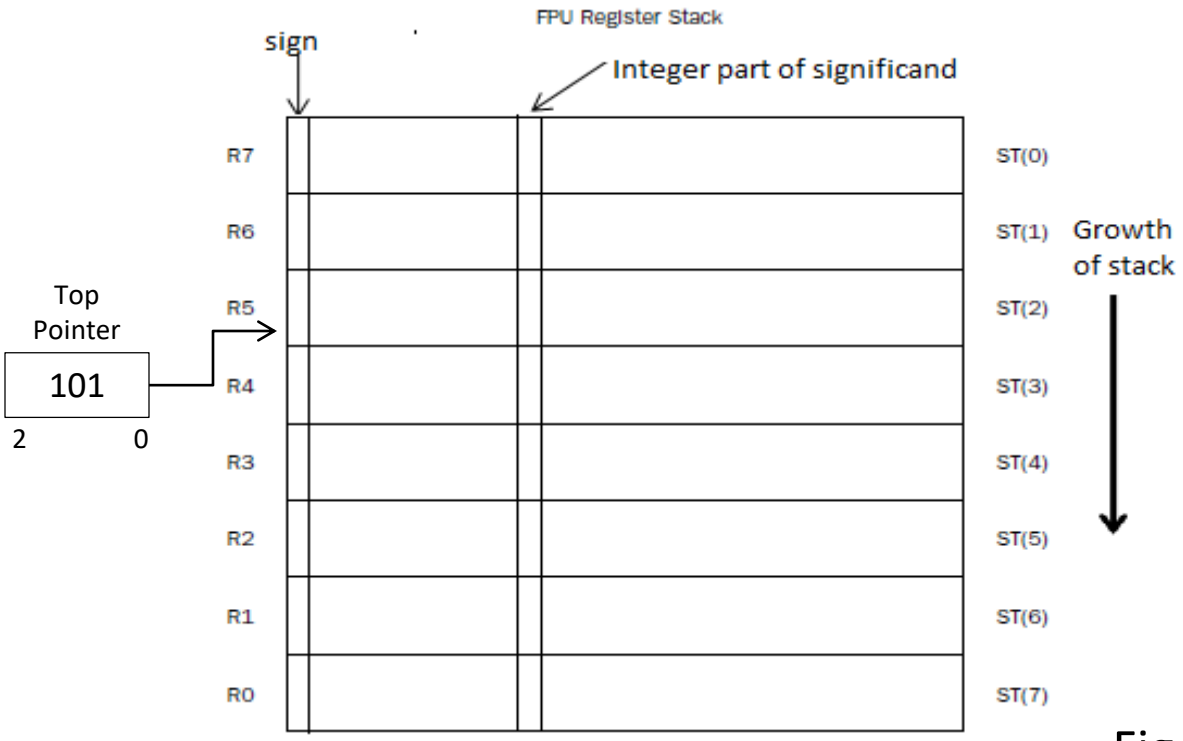


Fig. x87 Data registers in IA32 processor

- **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

- IA32 environment includes 8 general purpose data registers, each of 80 bits and capable to store real no in double extended precision format.

- In addition to general purpose data register, the x87 , operating environment includes three 16 bit registers .These registers are the following
  - X87 control register
  - X87 status register
  - X87 tag register

Architecture of floating point processor

- Control register:
  - Is used to control the way x87 instructions handle the precision and rounding off operands.
    - Bits for exception masking
    - Bits for computation control
  - 6 different floating point exceptions
  - Occurrence of these exceptions can be masked by setting the corresponding mask bit in the control register to 1

- Computation control bits are used to control precision and rounding .
- Infinity control bit for compatibility with the x87 FPUs. In GNU/Linux based computation it is always 0
- Rounding is needed when
  - When no in register is to be converted to a lower precision i.e. single or double

Architecture of floating point processor

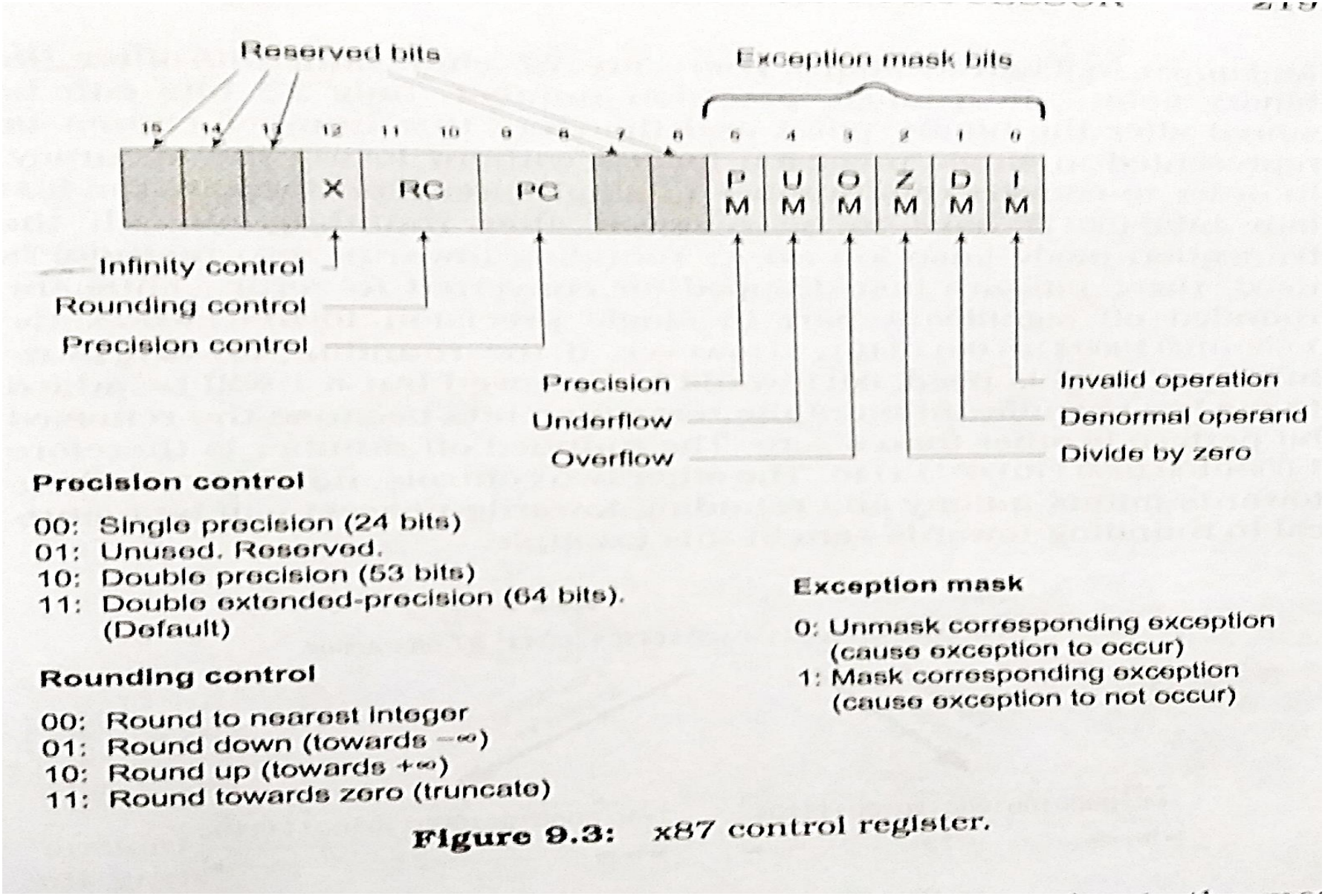


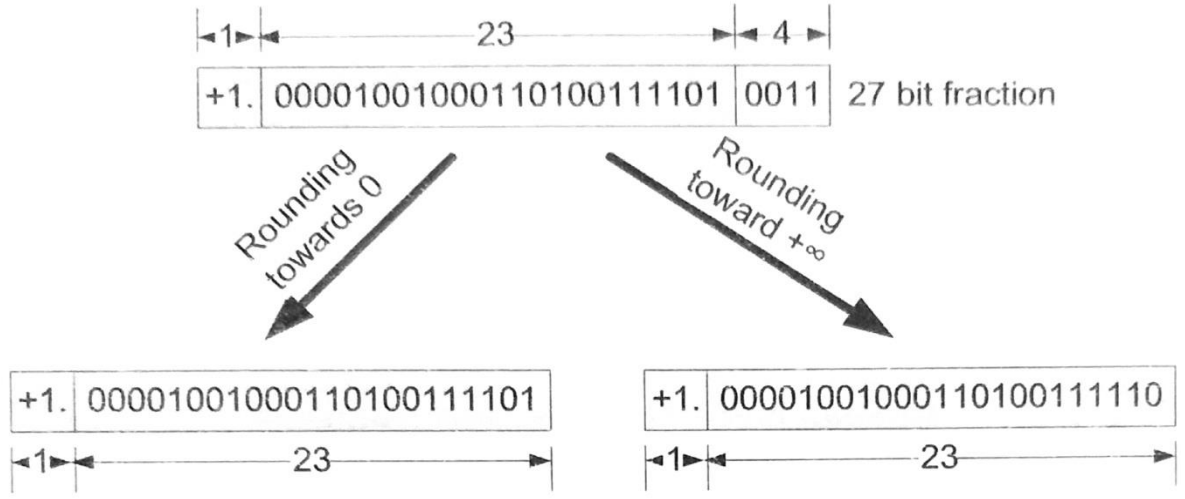
Figure 9.3: x87 control register.

- Used to control the precision and rounding of operands
- Infinity control bit is always set to 0

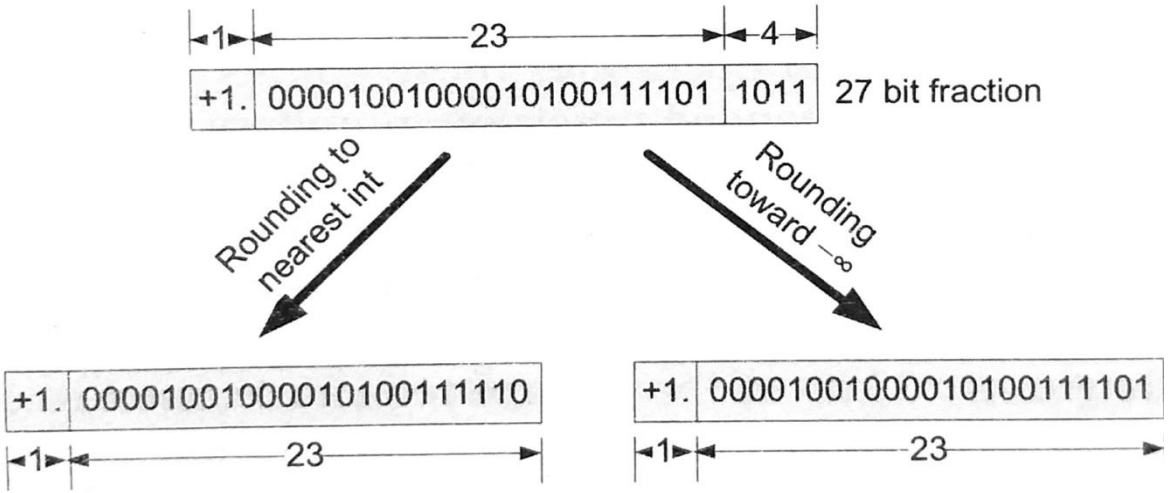
Control register

- X87 FPU provides 6 different kinds of FP exceptions
- Occurrence of these exceptions can be masked by setting the corresponding mask bit in the control register.
- **PC** bits specify precision(single or double or double extended) of FP computations
- **RC** bits used to define way of rounding is performed. (shown in fig 9.4)

Rounding control in IA32 :



(a) Rounding controls (towards 0 and +∞)



(b) Rounding controls (towards nearest and −∞)

Figure 9.4: Rounding controls in IA32 architectures.

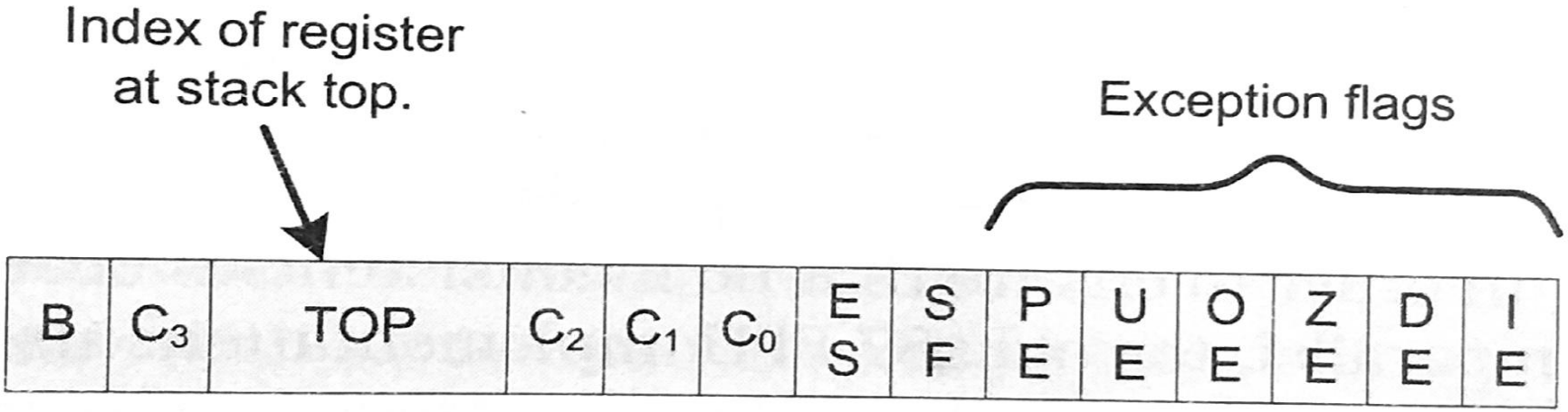
Used to handle the precision and rounding of operands

Infinity bit always set to 0

X87 status register:

- It indicates various exception flags, condition codes and stack top details.

X87 status register:



B: Busy flag
0: FPU free
1: FPU busy

$C_3..C_0$: Condition codes

ES: Error status
1: Error occurred
0: No error

SF: Stack fault

Exception flags
PE: Precision exception
UE: Underflow exception
OE: Overflow exception
ZE: Divide-by-zero exception
DE: Denormal operand exception
IE: Invalid operation exception

Figure 9.5: x87 status register.

- Status register can be saved in memory and can be taken into ax register.

- 4 condition code flags C_0 to C_3 –generated during execution of instruction
  - Theses indicates the result of floating point comparison and arithmetic operations.
  - Integer compare and branch instructions may be used to operate according to condition codes.

- **Exceptions flags** indicates various FP error conditions that may occur during execution of X87 instruction.

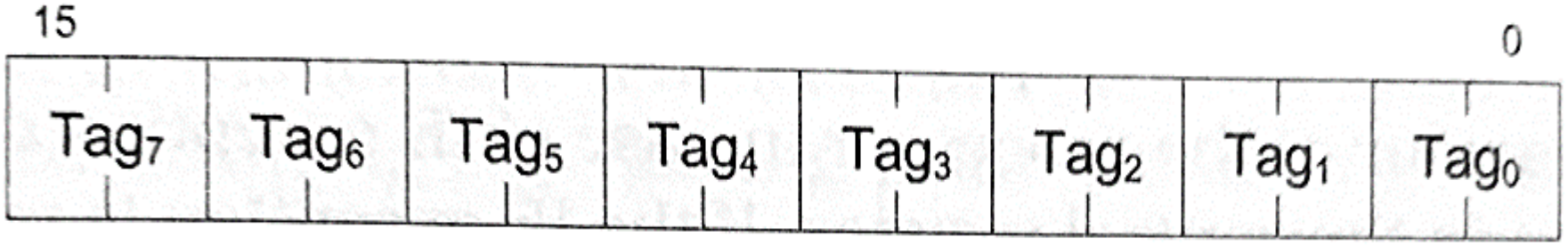
- If FP exception occurs during computation , ES (Error status) bit is set in the status register.
  - e. g. if an exception causes divide by zero FP exception , ZE flag is set to 1.

- **SF- Stack fault---** indicates errors due to stack overflow or underflow conditions.

- **B-Busy bit** ---for compatibility and most programs do not use this bit.

X87 data register:

- Each data register of X87 FPU has 2 bit tag contained in tag register.
- Code in the tag register indicates the type of the value of the **corresponding data register.**
- e.g. If Data register contain valid normalized value ,corresponding tag code is **00**



Tag_i is associated with data register Ri

00→ Ri has a valid value

01→ Ri has a zero

10→ Ri has a special value (NaN, ±∞ or denormalized number)

11→ Ri is empty (has not been loaded yet)

Figure 9.6: x87 tag register.

Floating point exceptions

1. **Invalid operation exception (IE)**
   - It is raised when operands of the instruction contain invalid values such as NaN.
   - 0*±infinity i.e. multiplication of 0 by ±∞ , division of ±∞ by ±∞
2. **Divide by zero exception (ZE)**
   - Occurs when divisor contains a zero and dividend is a finite number other than 0
3. **De-normalized operand exception (DE)**
   - It is raised when one of the operands of an instruction is in de-normalized form.
   - De-normalized numbers can be used to represent numbers with magnitudes too small to normalize (i.e. below 1.0×2^{-126})
4. **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.
5. **Numeric underflow exception (UE)**
   - Occur when the instruction generate result whose magnitude is less than the smallest possible normalized number.
6. **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.

Architecture of floating point processor

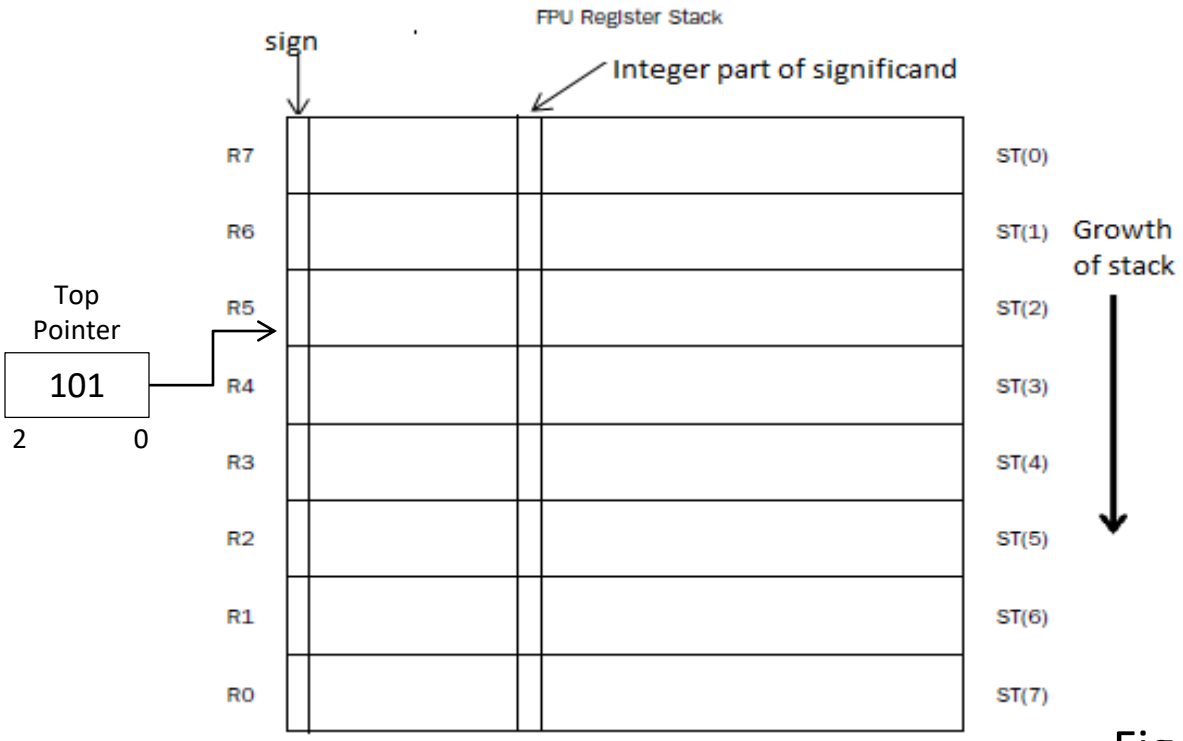


Fig. x87 Data registers in IA32 processor

- **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

- **X87 register addressing:**
- Register stack uses registers in a circular buffer
- Data register are organized as stack of registers
- e.g. if the top field in the x87 status register has a value 5 then **st(0), refers to x87 data register R5 as shown in Table 9.7**

Table 9.7: Register stack addressing in x87 FPU

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

Floating point instructions:

- Basic arithmetic instruction
- Constant loading instructions
- Trigonometric, logarithmic and exponentiation instructions
- Data comparison instructions
- Data transfer instructions
- FPU control instructions

Basic arithmetic instruction

- Addition instructions:

fadd memVar

faddl value1

: adds double precision FP no stored at memory location value1 to the stack top

fadd src, dest

fadd %st, %st(3)

: adds FP no in register st to register st(3), result is in st(3)

faddp

faddp

st(1)=st(1)+st(0), pop at st(0)

faddp dest

faddp %st(3)

st(3)=st(3)+st(0), pop the stack

,result will be available at st(2)(new TOS)

**fiadd memVar --- ** Add a 16- or 32-bit integer value to st(0) and store result at ST(0)

–Fiadds Value1

-**add 16 bit** integer(short int -suffix s) stored at memory location value1 to a floating point number in **st**

– Fiaddl Value2

-**add 32 bit (long int- suffix l)** integer stored at memory location value2 to a floating point number in st

Subtraction instruction:

fsubs/ fsubl no1----subtract single/double precision floating point (32 bit/64 bit) number **no1** from st(0) and store result at st(0)

st(0)= st(0)- no1

fsub st(3), st(0) ------ st(0)= st(0)- st(3)

fsubp st(3) ------ **st(3)= st(3)- st(0)**

:after the operation ,one item is removed from stack (pop)and hence st(3) becomes st(2)

fisubs/ fisubl val1 ---- Subtract a 16 or 32 bit integer value of memory from st(0) and store result at st(0)

st(0)= st(0) – 16/32 bit integer from memory location val1

Fisubs val1

st(0)= st(0) –val1

fsubr st(3),st(0) -------- st(0)= st(3)- st(0)

fsubrp st(2)-------- st(2)= st(0)- st(2)

:after the operation ,one item is removed from stack (pop)and hence st(2) becomes st(1)

fisubrl val1 ------- subtract st(0) from 32 bit int from memory location val1

st(0) = val1 - st(0)

Multiplication and division instructions:

Instruction	Description
fmuls val1	st(0)= st(0) * single precision floating point no stored at memory location val1
fmul st(3),st(0)	st(0)= st(0) *st(3)
fmulp st(6)	st(6)= st(6) *st(0)—after operation, stack is incremented (pop) by 1 and hence st(6) becomes st(5)
fimuls val1	st(0)= st(0) *16 bit integer from memory location val1

Multiplication and division instructions:

Instruction	Description
fdivl val1	st(0)= st(0) / double precision floating point no stored at memory location val1
fdiv st(0),st(3)	st(3)= st(3) *st(0)
fdivp	st(1)= st(1) /st(0)—after operation, stack is incremented (pop) by 1 and hence st(1) becomes st(0)
fidivl val32	st(0)= st(0) /32 bit integer from memory location val32
fdivrl val1	double precision floating point no stored at memory location **val1/st(0)**
fdivr st(3),st(0)	st(0)= st(3) /st(0)

Remainder computation:

Instruction	Description
fprem	➢Compute the remainder when st(0) is divided by st(1) and return remainder in st(0) ➢Truncating the Division result to an integer
fprem1	➢Compute the remainder when st(0) is divided by st(1) and return remainder in st(0) ➢Rounding the Division result to the nearest integer

#An example of basic FPU math to compute ((43.65 / 22) + (76.34 * 3.1)) / ((12.43 * 6) – (140.2 / 94.21))

#step-by-step analysis of what to perform the calculation

1. Load 43.65 into ST0.
2. Divide ST0 by 22, saving the results in ST0.
3. Load 76.34 in ST0 (the answer from step 2 moves to ST1).
4. Load 3.1 in ST0 (the value in step 3 moves to ST1, and the answer from Step 2 moves to ST2).
5. Multiply ST0 and ST1, leaving the answer in ST0.
6. Add ST0 and ST2, leaving the answer in ST0 (this is the left side of the equation).
7. Load 12.43 into ST0 (the answer from Step 6 moves to ST1).
8. Multiply ST0 by 6, leaving the answer in ST0.
9. Load 140.2 into ST0 (the answer from Step 8 moves to ST1, and from Step 6 to ST2).
10. Load 94.21 into ST0 (the answer from Step 8 moves to ST2, and from Step 6 to ST3).
11. Divide ST1 by ST0, popping the stack and saving the results in ST0 (the answer from Step 8 moves to ST1, and from Step 6 to ST2).
12. Subtract ST0 from ST1, storing the result in ST0 (this is the right side of the equation).
13. Divide ST2 by ST0, storing the result in ST0 (this is the final answer)

```
.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
```

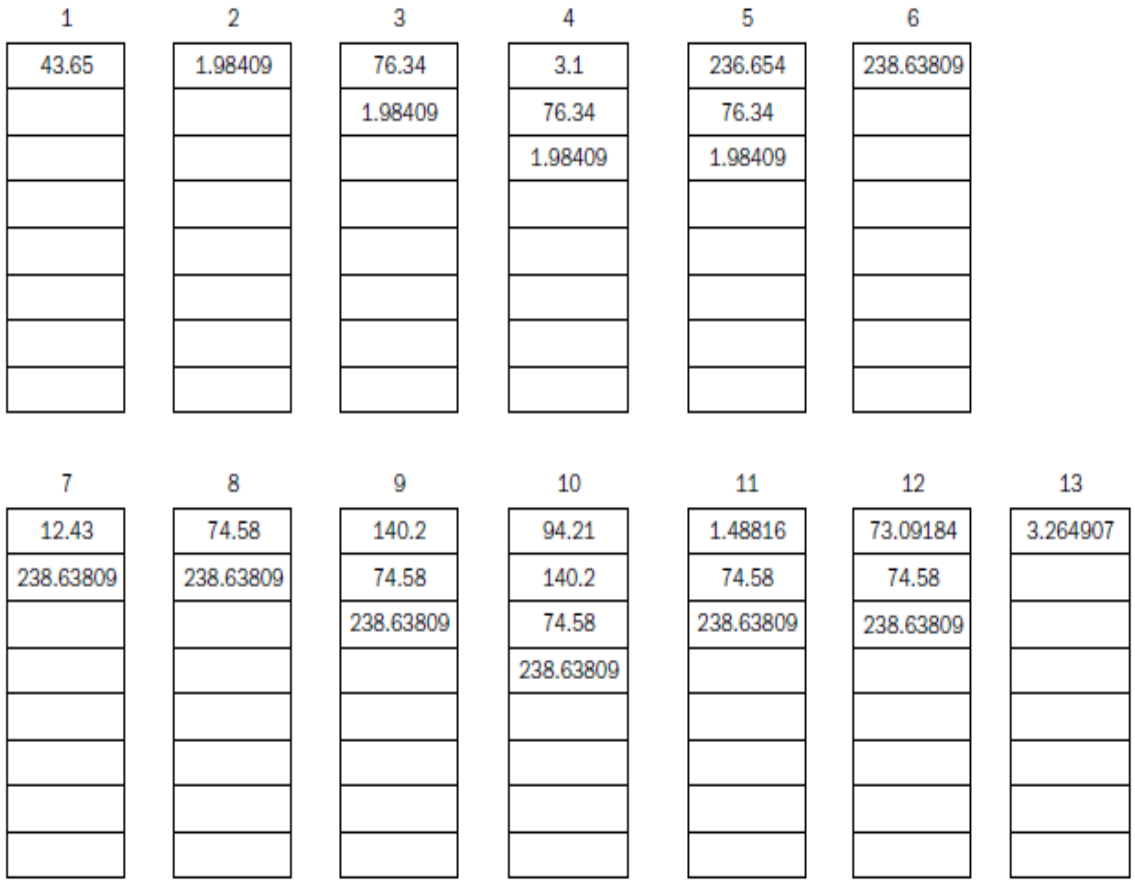


Fig. Calculation sequence

Other instructions:

Instruction	Description
Square root computation	
fsqrt	➢ Square root of a number on top of stack st(0) is computed using **fsqrt** ➢ result is in st(0)
Miscellaneous computations	
fchs	➢Changes the sign of the input on the top of stack st(0) ➢ After execution +ve no become –ve and –ve no becomes +ve.
fscale	➢ takes 2 operands on the stack (i.e. st(0) and st(1) and replaces the TOS by the result of computation - $\mathbf{st(0) = st(0) * 2^{st(1)}}$
fabs	Compute the absolute value of register st(0)

#Example of the FABS, FCHS, and FSQRT instructions

```
.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
```

#After execution result is :

```
(gdb) info all
.
.
.
st0   8
st1    9145.2900390625
st2    -395.209991455078125

(gdb)
```

#An example of the FSCALE instruction

```
.section .data
value:
        .float 10.0
scale1:
        .float 2.0
scale2:
        .float -2.0
.section .bss
  .lcomm result1, 4
   .lcomm result2, 4
```

```
.section .text
.globl _start
_start:
Nop
finit
flds scale1
flds value
fscale
fsts result1
flds scale2
flds value
fscale
fsts result2
movl $1, %eax
movl $0, %ebx
int $0x80
```

After execution result is:

```
(gdb) x/f  &result1
0x80490b8 <result1>: 40
(gdb) x/f  &result2
0x80490bc <result2>: 2.5
(gdb)
```

#finding Roots of a quadratic equation

Val1=4 ---int

val2=2 --int

a=1 --float

b= -4 --float

C=3 --float

Val3=0.0 and val4=0.0 ---float

$$x1, x2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

#finding Roots of a quadratic equation

```
.section .data
a:
.float 1
b:
.float -4
c:
.float 3
val1:
.float 4
val2:
.float 2
val3:
.float 0.0
val4:
.float 0.0
val5:
.float 0.0
```

```
.section .text
.globl _start
_start:
    nop
    flds a
    flds c
    fmul %st(1),%st(0)
    fmul val1
    fsts val3
    flds b
    fmul %st(0),%st(0)
    fsub %st(1),%st(0)
    fsqrt
    fsts val4
    flds b
    fadd %st(1),%st(0)
    flds val2
    fmul %st(4),%st(0)
```

```
fsts val5
fdiv %st(1),%st(0)
flds b
flds val4
fsubr %st(0),%st(1
flds val5
fdivr %st(2),%st(0

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

print $st0

or

Info all

Constant loading instructions:

- These instructions are used to load constants on the top of the stack(TOS)

Instruction	Description
fld1	Loads constant +1.0
fldz	Loads constant +0.0
fldpi	Loads constant ∏ (pi)
fldl2e	Loads constant $\log_2 e$
fldln2	Loads constant $\log_e 2$
fldl2t	Loads constant $\log_2 10$
fldlg2	Loads constant $\log_{10} 2$

Trigonometric log and exponentiation instructions:

- These instructions compute Trigonometric functions for arguments stored on the top of the stack(TOS)

Instructions	Description
fsin	Compute sine of an angle in radians provided in st(0) and result is in st(0)
fcos	Compute cosine of an angle in radians provided in st(0) and result is in st(0)
fsincos	Compute sine and cosine of an angle in radians provided in st(0) --st(0) contains cosine and st(1) contains sine of the angle
fptan	Computes tangent of the angle
fpatan	Computes arctangent of the angle

- #10 Write assembly language program in 80386 to convert degree to radian and compute its sin, cosine and tan value.

#degree to radian conversion and calculating the angle

radians = (degrees * pi) / 180

```
.section .data
   degree1:
      .float 60.0
   val180:
      .int 180
   .section .bss
      .lcomm rad1, 4
      .lcomm out1, 4
      .lcomm out2, 4
   .section .text
   .globl _start
   _start:
      nop
      finit
        flds degree1
        fidivl val180
        fldpi
        fmul %st(1), %st(0)
        fsts rad1
        fsin
        fsts out1 ---- (0.8660)
        flds rad1
        fcos
        fsts out2
   movl $1, %eax
        movl $0, %ebx
        int $0x80
```

Logarithmic instructions:

- These instructions set to compute logarithm.

Instructions	Description
fyl2x	➢ It takes 2 arguments x and y on the stack st(0) and st(1) resp. ➢ Compute the $y.log_2x$
fyl2xp1	➢ It takes 2 arguments x and y on the stack st(0) and st(1) resp. ➢ Compute the $y.log_2(x+1)$

#example of using the FYL2X instruction

To find a logarithm of another base using base 2 logarithms, we can use the equation:

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

```
.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
```

```
#after execution result is-
(gdb) x/f &result
0x80490a8 <result>: 1.07918119
(gdb)
```

Exponentiation instruction:

- This instructions computes 2^x-1

Instruction	Description
f2xm1	Result is at top of the stack st(0)

FPU control instructions:

Instruction	Description
finit	initialize the x87 FPU
fninit	It must be executed when the MMX computing environment is changed to x87 FPU computing environment

- These instructions initialize the x87 FPU state to the following:
  - **Control register** is initialized to 0x37f ,which masks all FP exceptions ,
  - set rounding control to round to nearest and
  - Precision control to double extended precision.
  - **status word** is set to 0 , meaning that no exceptions are set,
  - All condition flags are set to 0 and
  - Register stack top s initialized to 0
  - **Tag word** is set to 0xFFFF , which makes all data registers marked as empty

program to get the FPU Status register contents

```
.section .bss
.lcomm status, 2
.section .text
.globl _start
_start:
nop
#fstsw %ax
fstsw status
movl $1, %eax
movl $0, %ebx
int $0x80
```

After execution

```
(gdb) x/x  &status
0x804908c <status>: 0x00000000
```

Or

```
(gdb) info all
.
.
.
fctrl 0x37f 895
fstat 0x0 0
ftag 0x55555 349525     #shows current values of 3 regis.
(gdb)
```

Or

```
(gdb) print/x   $fstat
$1 = 0x0
(gdb)  print/x   $ftag
$2 = 0xffff
(gdb)  print/x   $fctrl
$3 = 0x37f
```

Data comparison instructions

- The FCOM family of instruction is used to compare two floating-point values in the FPU.

- The instructions compare the value loaded in the ST0 FPU register with either another FPU register or a floating point value in memory.

Instruction	Description
FCOM	Compare the ST0 register with the ST1 register.
FCOM ST(x)	Compare the ST0 register with another FPU register.
FCOM source	Compare the ST0 register with a 32- or 64-bit memory value.
FCOMP	Compare the ST0 register with the ST1 register value and pop the stack.
FCOMP ST(x)	Compare the ST0 register with another FPU register value and pop the stack.
FCOMP source	Compare the ST0 register with a 32 or 64-bit memory value and pop the stack.
FCOMPP	Compare the ST0 register with the ST1 register and pop the stack twice.
FTST	Compare the ST0 register with the value 0.0.

The result of the comparison is set in the C0, C2, and C3 condition code bits of the status register.

Condition	C3	C2	C0
ST0 > source	0	0	0
ST0 < source	0	0	1
ST0 = source	1	0	0

- For single precision floating point no stored in memory then
  - Use fcoms and fcomps
- For double precision floating point no stored in memory then
  - Use fcoml and fcompl
  - Fcom %st(3) ---- compares stack top with st(3)

FCOMI instruction

- The FCOMI family of instructions performs the floating-point comparisons and places the results in the **EFLAGS registers** using the carry, parity, and zero flags.

Instruction	Description
FCOMI	Compare the ST0 register with the ST(x) register.
FCOMIP	Compare the ST0 register with the ST(x) register and pop the stack.
FUCOMI	Check for unordered values before the comparison.
FUCOMIP	Check for unordered values before the comparison and pop the stack afterward.

Condition	ZF	PF	CF
ST0 > ST(x)	0	0	0
ST0 < ST(x)	0	0	1
ST0 = ST(x)	1	0	0

FCOMI src, %st

FCOMIP src, %st

FUCOMI src, %st

FUCOMIP src, %st

Exchanging values

- Fxchg src
  - Top of stack exchanged with any of src i.e st(i)
- Fxchg
  - Top of stack exchanged with st(1)

SIMD Introduction:

- Intel introduced multimedia extension (MMX) instruction set with Pentium processors.

- MMX was the first technology to support the Intel **Single Instruction, Multiple Data (SIMD) execution model.**

- Most Instructions operate simultaneously on multiple data values.

- The SIMD model was developed to process larger numbers, commonly found in applications such **as image processing , multimedia applications including voice and data communication.**

- With Pentium III, Intel introduced **streaming SIMD extension** (SSE) instruction set.

- **MMX** instructions operate on **integer data**, and **SSE** instructions operate on **floating point data.**

- SSE enhances performance for complex floating-point arithmetic, often used in 3-D graphics, motion video, and video conferencing.

- SSE2 – second enhancement to the SSE with 128 bit wide registers and operations on multiple integers and floating point nos.

SIMD environment

- The Intel Single Instruction Multiple Data (SIMD) technology provides additional ways to define integers.
- These integer perform arithmetic operations on a group of multiple integers simultaneously.
- The SIMD architecture uses the packed integer data type.
- A packed integer is a series of bytes that can represent more than one integer value.
- SIMD instruction set (i.e. MMX,SSE and SSE2) provide a few additional register called **MMX registers and XMM** registers.
- **MMX registers includes 8,64 bit registers named mm0-mm7.**

SIMD Integers and SIMD environment

- **Data register of x87 FPU are 80 bit wide** and capable of storing double extended precision FP numbers. Out of these 80 bits only 64 bits are used by instructions in the SIMD instruction set.

- Use **fninit or 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**.
  - i.e. XMM registers stores four, single precision floating point nos or
  - two, single precision floating point nos or
  - Multiple integers in byte, word, long word or single 128 bit integer format.

- XMM registers are independent registers and are not shared with x87 FPU data registers unlike MMX registers.

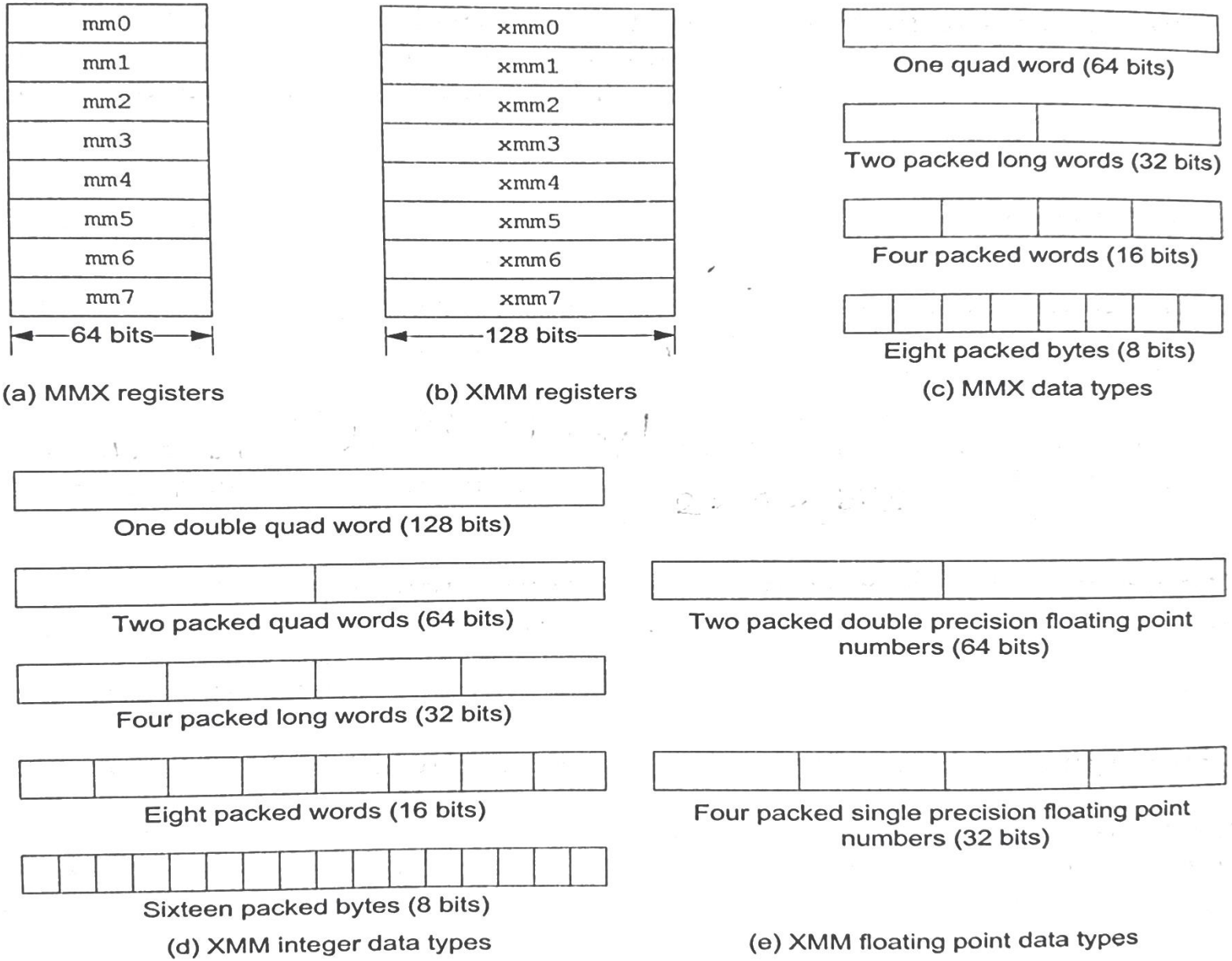


Figure 10.1: MMX registers and supported data types.

SIMD Integers:

- The Intel Single Instruction Multiple Data (SIMD) technology provides additional ways to define integers

- These new integer types enable the processor to perform arithmetic operations on a group of multiple integers simultaneously.

- The SIMD architecture uses the packed integer data type.

- A packed integer is a series of bytes that can represent more than one integer value.
  - Mathematical operations can be performed on the series of bytes as a whole

- **movq source, destination ---**move data into MMX register
  - source and destination can be an MMX register, an SSE register, or a 64-bit memory location

#Loading and retrieving packed integer values

```
.section .data
packedvalue1:
        .byte 10, 20, -30, 40, 50, 60, -70, 80          #8-byte integer values
packedvalue2:
        .short 10, 20, 30, 40                           #four word integer values
packedvalue3:
        .int 10, 20                                     #two double word integer values.
.section .text
.globl _start
_start:
        movq packedvalue1, %mm0
        movq packedvalue2, %mm1
        movq packedvalue3, %mm2
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

print $mm0
print $mm1
print $mm2

Or reserve memory
in the .bss section

Performing MMX operations

- Once the data is loaded into the MMX register, parallel operations can be performed on the packed data using a single instruction.
- The operations are performed on each packed integer value in the register, utilizing the same placed packed integer values. As shown in following **fig a**

- ***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.

- With packed integer values, multiple result values are computed simultaneously. This means that a single set of flags cannot indicate the result of the operation, as in normal integer math.

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

- There are three ways to handle the conditions of result overflow or underflow so the result will be meaningful:
  - Normal operation (Wraparound arithmetic)
  - Operation with Signed saturation
  - Operation with Unsigned saturation

- Overflow flag – to represent signed number overflow
- carry flag – to represent unsigned number overflow
- To indicate that the result cannot be represented in designed no of bits.

Data Type	Positive Overflow Value	Negative Overflow Value
Signed byte	127	-128
Signed word	32,767	-32,768
Unsigned byte	255	0
Unsigned word	65,535	0

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
0x44	0x23	A-B	0x21	0x21	0x21

Wrap around- Whenever the result is larger than 0xFF, Extra bit is dropped. And wrap around results is 0x65

- **Saturation modes** ensures that results are not outside the range of minimum and maximum possible values.
  - i.e. while using signed saturation
    - when the true result of an operation is smaller than the minimum negative number , the result is set to the minimum negative number.
    - Similarly the result is set to the maximum positive number when the true result is more than the maximum positive number.

- While using unsigned saturation
  - The results are set to 0 if the true result is negative and
  - set to maximum positive value if the true result is larger than the maximum possible value.

- **Signed saturation-** when -93 and -62 in decimal added. The summation of two numbers will be -155 which is below the minimum possible and cannot be represented in 8 bits. And provides results as 0x80 (i.e. -128)

- **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 (i.e. 255)

SIMD integer arithmetic instructions:

- Operations that use MMX registers were introduced with MMX instruction set.
- Operations that use XMM registers were introduced with SSE2 instruction set.

Instruction	*Operation*	*Inst. sets*
	Addition instructions	
`paddb src, dest`	Add packed bytes	MMX, SSE2
`paddw src, dest`	Add packed words	MMX, SSE2
`paddd src, dest`	Add packed longs (or doublewords)	MMX, SSE2
`paddq src, dest`	Add packed quadwords	SSE2
`paddsb src, dest`	Add packed bytes with signed saturation	MMX, SSE2
`paddsw src, dest`	Add packed words with signed saturation	MMX, SSE2
`paddusb src, dest`	Add packed bytes with unsigned saturation	MMX, SSE2
`paddusw src, dest`	Add packed words with unsigned saturation	MMX, SSE2

These instructions operate on 64 bit packed data stored in MMX registers or on 128 bit packed data stored in XMM registers.

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
```

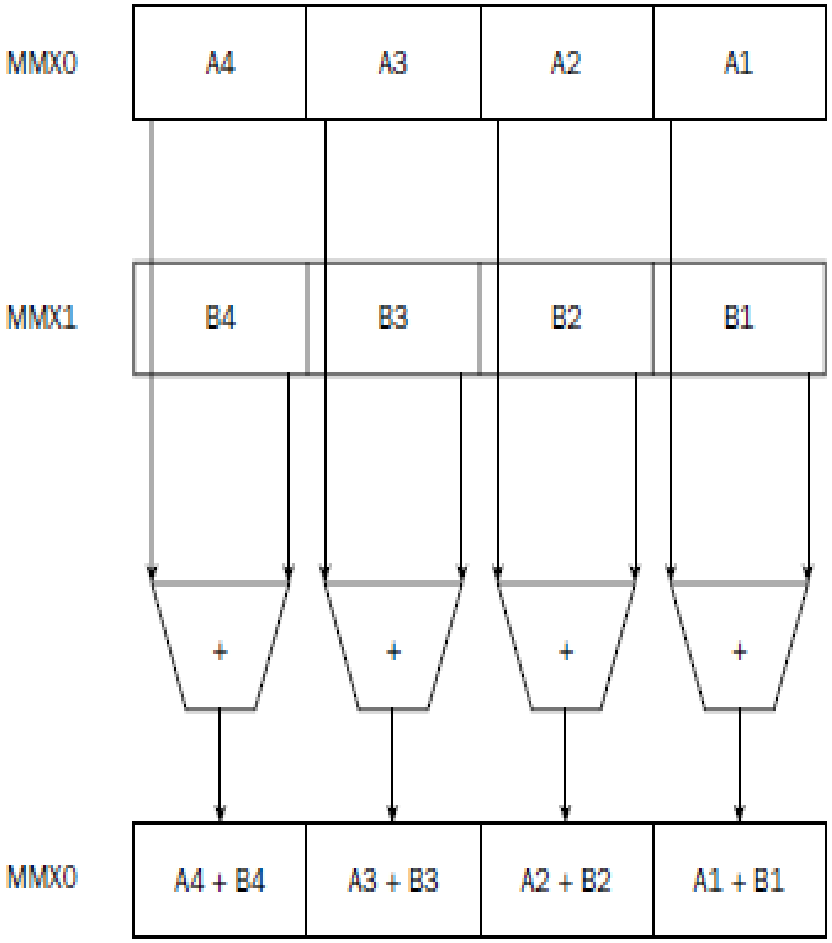


Fig a

```
(gdb) x/2d &value1
0x804909c <value1>:     10      20
(gdb) x/2d &value2
0x80490a4 <value2>:     30      40
(gdb) x/2d &result
0x80490b0 <result>:     40      60
(gdb)
```

MMX math operations

MMX Instruction	Description
PADDB	Add packed byte integers with wraparound
PADDW	Add packed word integers with wraparound
PADDD	Add packed doubleword integers with wraparound
PADDSB	Add packed byte integers with signed saturation
PADDSW	Add packed word integers with signed saturation
PADDUSB	Add packed byte integers with unsigned saturation
PADDUSW	Add packed word integers with unsigned saturation
PSUBB	Subtract packed byte integers with wraparound
PSUBW	Subtract packed word integers with wraparound
PSUBD	Subtract packed doubleword integers with wraparound
PSUBSB	Subtract packed byte integers with signed saturation
PSUBSW	Subtract packed word integers with signed saturation
PSUBUSB	Subtract packed byte integers with unsigned saturation
PSUBUSW	Subtract packed word integers with unsigned saturation

Using SSE Instructions:

- The SSE architecture provides SIMD support for packed single-precision floating-point values.

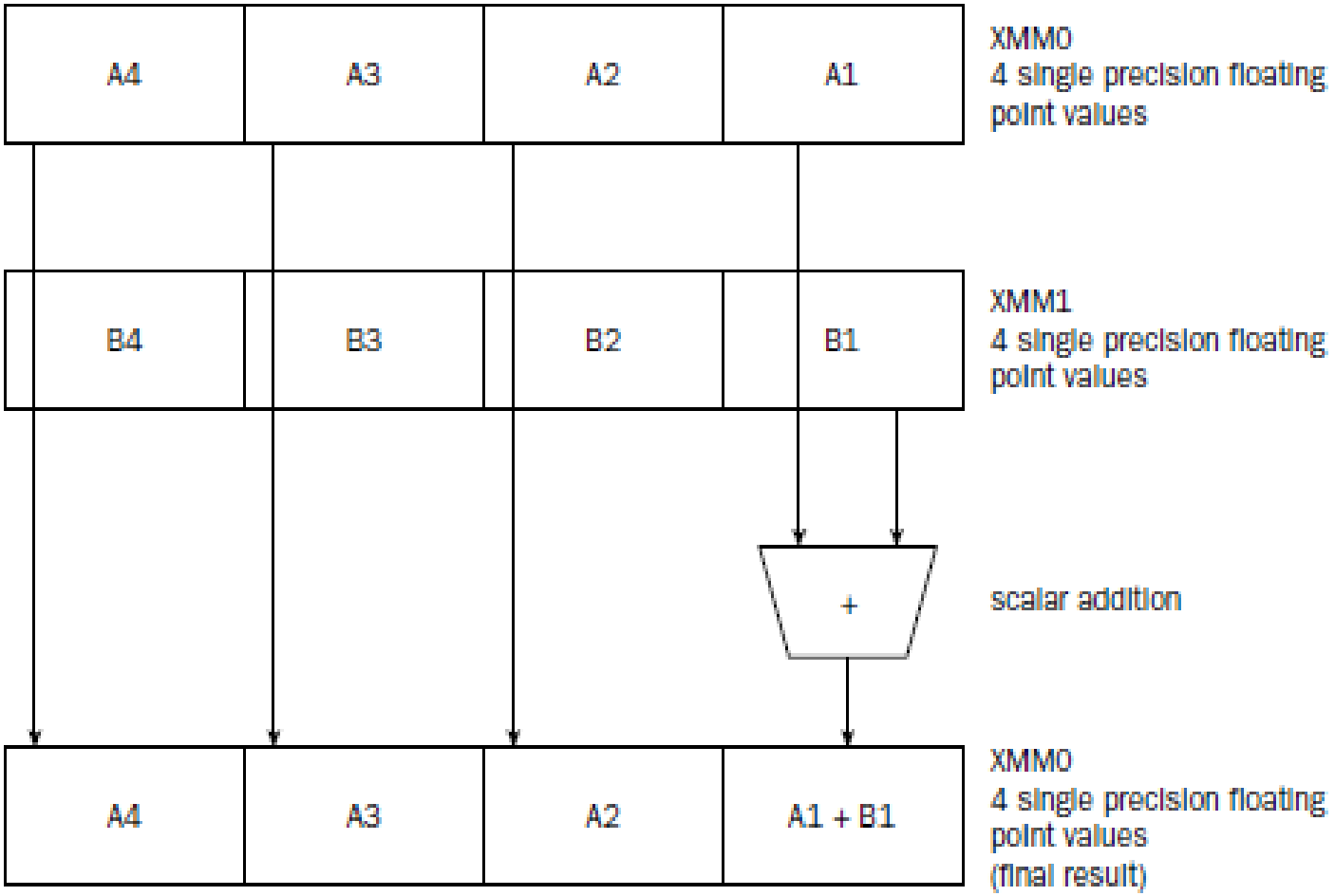
- SSE provides new instructions for moving data into XMM registers, processing mathematical operations on the SSE data, and retrieving data from the XMM registers.

- SSE instruction has two versions.
- The first version uses a PS suffix.
  - These instructions perform the arithmetic operation on the packed single-precision floating-point value similarly to how MMX operates.

- The second version of the arithmetic instruction uses an SS suffix.
  - These instructions perform the arithmetic operation on a scalar single-precision floating-point value.
  - Instead of performing the operation on all of the floating-point values in the packed value, it is performed on only the low doubleword in the packed value. The remaining three values from the source operand are carried through to the result, as shown in following fig.

SSE Instructions

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



- The scalar operations enable normal FPU-type arithmetic operations to be performed on one single precision floating-point value in the XMM registers.

Moving Data in 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.

- **MOVAPS : This instruction** move four aligned, single-precision values to XMM registers or memory.
- The SSE MOVAPS instruction expects the data to be located on a 16-byte memory boundary

This makes it easier for the processor to read the data in a single operation.

SSE Arithmetic instructions

Instruction	Description
ADDPS	Add two packed values.
SUBPS	Subtract two packed values.
MULPS	Multiply two packed values.
DIVPS	Divide two packed values.
RCPPS	Compute the reciprocal of a packed value.
SQRTPS	Compute the square root of a packed value.
RSQRTPS	Compute the reciprocal square root of a packed value.
MAXPS	Compute the maximum values in two packed values.
MINPS	Compute the minimum values in two packed values.
ANDPS	Compute the bitwise logical AND of two packed values.
ANDNPS	Compute the bitwise logical AND NOT of two packed values.
ORPS	Compute the bitwise logical OR of two packed values.
XORPS	Compute the bitwise logical exclusive-OR of two packed values.

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
```

```
Breakpoint 1, _start () at ssemath.s:14
14          movaps value1, %xmm0
Current language:  auto; currently asm
(gdb) s
15          movaps value2, %xmm1
(gdb) s
17          addps %xmm1, %xmm0
(gdb) print $xmm0
$1 = {f = {12.3400002, 2345, -93.1999969, 10.4399996}}
(gdb) print $xmm1
$2 = {f = {39.2340012, 21.3999996, 100.940002, 10.5600004}}
```

```
(gdb) s

(gdb) print $xmm0
$3 = {f = {51.5740013, 2366.3999, 7.74000549, 21}}
```

or

x/4f &result

Thank You