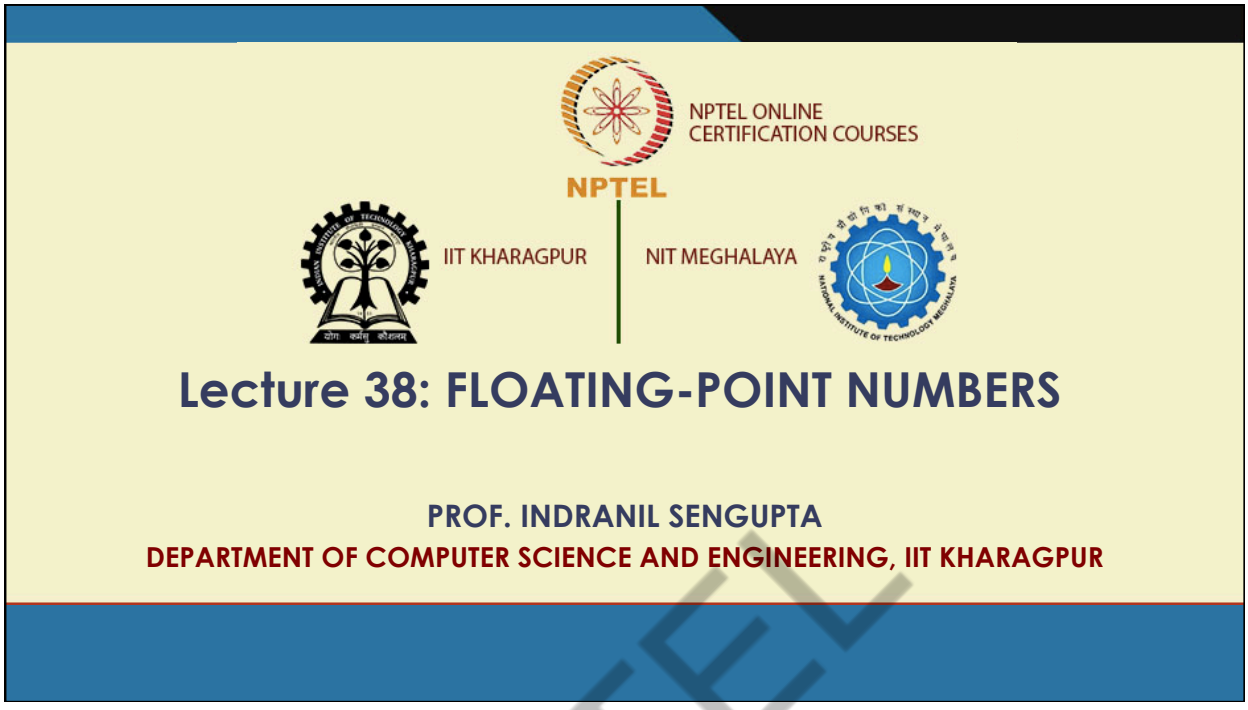


Representing Fractional Numbers

- A binary number with fractional part

  $B = b_{n-1}\, b_{n-2} \ldots\ldots b_1\, b_0 \bullet b_{-1}\, b_{-2} \ldots\ldots b_{-m}$

  corresponds to the decimal number

  $D = \sum_{i=-m}^{n-1} b_i\, 2^i$

If the radix point is allowed to move, we call it a floating-point representation.

- Also called *fixed-point numbers*.
  - The position of the radix point is fixed.

Some Examples

1011.1 ➔ $1x2^3 + 0x2^2 + 1x2^1 + 1x2^0 + 1x2^{-1}$ = 11.5

101.11 ➔ $1x2^2 + 0x2^1 + 1x2^0 + 1x2^{-1} + 1x2^{-2}$ = 5.75

10.111 ➔ $1x2^1 + 0x2^0 + 1x2^{-1} + 1x2^{-2} + 1x2^{-3}$ = 2.875

Some Observations:

- Shift right by 1 bit means divide by 2
- Shift left by 1 bit means multiply by 2
- Numbers of the form $0.111111\ldots_2$ has a value less than 1.0 (one).

Limitations of Representation

- In the fractional part, we can only represent numbers of the form $x/2^k$ exactly.
  - Other numbers have repeating bit representations (i.e. never converge).
- Examples:

  3/4 = 0.11
  7/8 = 0.111
  5/8 = 0.101
  1/3 = 0.10101010101 [01] ….
  1/5 = 0.001100110011 [0011] ….
  1/10 = 0.0001100110011 [0011] ….

- More the number of bits, more accurate is the representation.
- We sometimes see: (1/3)*3 ≠ 1.

Floating-Point Number Representation (IEEE-754)

- For representing numbers with fractional parts, we can assume that the fractional point is somewhere in between the number (say, *n* bits in integer part, *m* bits in fraction part). → *Fixed-point representation*
  - Lacks flexibility.
  - Cannot be used to represent very small or very large numbers (for example: 2.53×10^{-26}, $1.7562 \times 10^{+35}$, etc.).
- Solution :: use floating-point number representation.
  - A number F is represented as a triplet *<s, M, E>* such that

    $F = (-1)^s M \times 2^E$

$F = (-1)^s M \times 2^E$

- s is the *sign bit* indicating whether the number is negative (=1) or positive (=0).
- M is called the *mantissa*, and is normally a fraction in the range [1.0,2.0].
- E is called the *exponent*, which weights the number by power of 2.

Encoding:

- Single-precision numbers: total 32 bits, E 8 bits, M 23 bits
- Double-precision numbers: total 64 bits, E 11 bits, M 52 bits

Points to Note

- The number of *significant digits* depends on the number of bits in M.
  - 7 significant digits for 24-bit mantissa (23 bits + 1 implied bit).
- The *range* of the number depends on the number of bits in E.
  - 10^{38} to 10^{-38} for 8-bit exponent.

How many significant digits?

$2^{24} = 10^x$

$24 \log_{10}2 = x \log_{10}10$

$x = 7.2$ -- 7 significant decimal places

Range of exponent?

$2^{127} = 10^y$

$127 \log_{10}2 = y \log_{10}10$

$y = 38.1$ -- maximum exponent value 38 (in decimal)

"Normalized" Representation

- We shall now see how *E* and *M* are actually encoded.
- Assume that the actual exponent of the number is *EXP* (i.e. number is $M \times 2^{EXP}$).
- Permissible range of *E*: $1 \leq E \leq 254$ (the all-0 and all-1 patterns are not allowed).
- **Encoding of the exponent E**:
  - The exponent is encoded as a biased value: $E = EXP + BIAS$

    where BIAS = 127 $(2^{8-1} - 1)$ for single-precision, and

    BIAS = 1023 $(2^{11-1} - 1)$ for double-precision.

- **Encoding of the mantissa M:**
  - The mantissa is coded with an implied leading 1 (i.e. in 24 bits).

    M = 1 . xxxx...x
  - Here, xxxx…x denotes the bits that are actually stored for the mantissa. We get the extra leading bit for *free*.
  - When xxxx…x = 0000…0, M is minimum (= 1.0).
  - When xxxx…x = 1111…1, M is maximum (= 2.0 – ε).

An Encoding Example

- Consider the number F = 15335

  $15335_{10} = 11101111100111_2 = 1.1101111100111 \times 2^{13}$

- Mantissa will be stored as: M = $1101111100111\ 0000000000_2$

- Here, EXP = 13, BIAS = 127. ➔ E = 13 + 127 = 140 = 10001100_2

0	10001100	11011111001110000000000

466F9C00 in hex

Another Encoding Example

- Consider the number F = -3.75

  $-3.75_{10} = -11.11_2 = -1.111 \times 2^1$

- Mantissa will be stored as: $M = 11100000000000000000000_2$

- Here, EXP = 1, BIAS = 127. ➔ $E = 1 + 127 = 128 = 10000000_2$

1	10000000	11100000000000000000000

40700000 in hex

Special Values

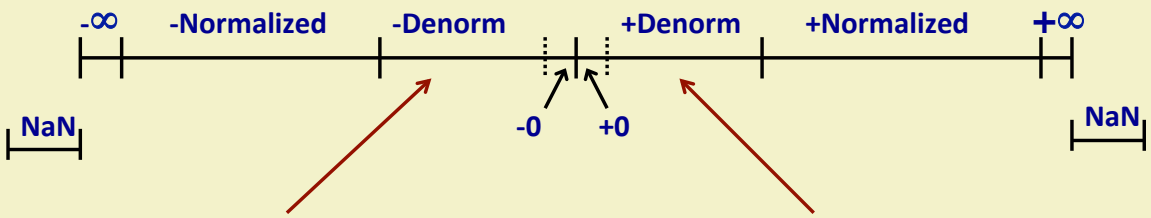
- When E = 000…0
  - M = 000…0 represents the value 0.
  - M ≠ 000…0 represents numbers very close to 0.
- When E = 111…1
  - M = 000…0 represents the value ∞ (infinity).
  - M ≠ 000…0 represents *Not-a-Number* (NaN).

Zero is represented by the all-zero string.

Also referred to as *de-normalized* numbers.

NaN represents cases when no numeric value can be determined, like uninitialized values, ∞*0, ∞-∞, square root of a negative number, etc.

Summary of Number Encodings



Denormal numbers have very small magnitudes (close to 0) such that trying to normalize them will lead to an exponent that is below the minimum possible value.

- Mantissa with leading 0's and exponent field equal to zero.
- Number of significant digits gets reduced in the process.

Rounding

- Suppose we are adding two numbers (say, in single-precision).
  - We add the mantissa values after shifting one of them right for exponent alignment.
  - We take the first 23 bits of the sum, and discard the residue R (beyond 32 bits).
- IEEE-754 format supports four rounding modes:
  a) Truncation
  b) Round to +∞ (similar to ceiling function)
  c) Round to -∞ (similar to floor function)
  d) Round to nearest

- To implement rounding, two temporary bits are maintained:
  - *Round Bit (r)*: This is equal to the MSB of the residue *R*.
  - *Sticky Bit (s)*: This the logical OR of the rest of the bits of the residue *R*.
- Decisions regarding rounding can be taken based on these bits:
  a) R > 0: If r + s = 1
  b) R = 0.5: If r.s' = 1
  c) R > 0.5: If r.s = 1 // '+' is logical OR, '.' is logical AND
- Renormalization after Rounding:
  - If the process of rounding generates a result that is not in normalized form, then we need to re-normalize the result.

Some Exercises

1. Decode the following single-precision floating-point numbers.
   a) 0011 1111 1000 0000 0000 0000 0000 0000
   b) 0100 0000 0110 0000 0000 0000 0000 0000
   c) 0100 1111 1101 0000 0000 0000 0000 0000
   d) 1000 0000 0000 0000 0000 0000 0000 0000
   e) 0111 1111 1000 0000 0000 0000 0000 0000
   f) 0111 1111 1101 0101 0101 0101 0101 0101

END OF LECTURE 38

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

17

NPTEL ONLINE CERTIFICATION COURSES

IIT KHARAGPUR | NIT MEGHALAYA

Lecture 39: FLOATING-POINT ARITHMETIC

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, IIT KHARAGPUR

Floating Point Addition/Subtraction

- Two numbers: $M1 \times 2^{E1}$ and $M2 \times 2^{E2}$, where $E1 > E2$ (say).
- Basic steps:
  - Select the number with the smaller exponent (i.e. *E2*) and shift its mantissa right by (*E1-E2*) positions.
  - Set the exponent of the result equal to the larger exponent (i.e. *E1*).
  - Carry out $M1 \pm M2$, and determine the sign of the result.
  - Normalize the resulting value, if necessary.

Addition Example

- Suppose we want to add F1 = 270.75 and F2 = 2.375

  $F1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$

  $F2 = (2.375)_{10} = (10.011)_2 = 1.0011 \times 2^1$
- Shift the mantissa of F2 right by 8 – 1 = 7 positions, and add:

  1000 0111 0110 0000 0000 0000

  1 0011 0000 0000 0000 0000 000

  1000 1000 1001 0000 0000 0000 0000 000 ← Residue
- Result: 1.00010001001×2^8

Subtraction Example

- Suppose we want to subtract F2 = 224 from F1 = 270.75

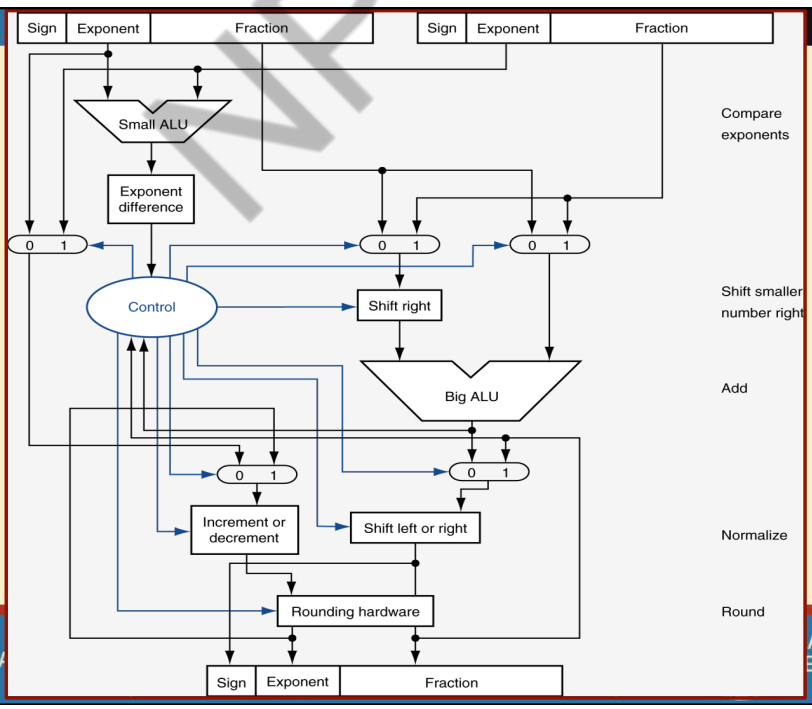
  $F1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$

  $F2 = (224)_{10} = (11100000)_2 = 1.111 \times 2^7$

- Shift the mantissa of F2 right by 8 – 7 = 1 position, and subtract:

  1000 0111 0110 0000 0000 0000
  111 0000 0000 0000 0000 0000 000
  0001 0111 0110 0000 0000 0000 000

- For normalization, shift mantissa left 3 positions, and decrement E by 3.
- Result: 1.01110110×2^5



Floating-Point Multiplication

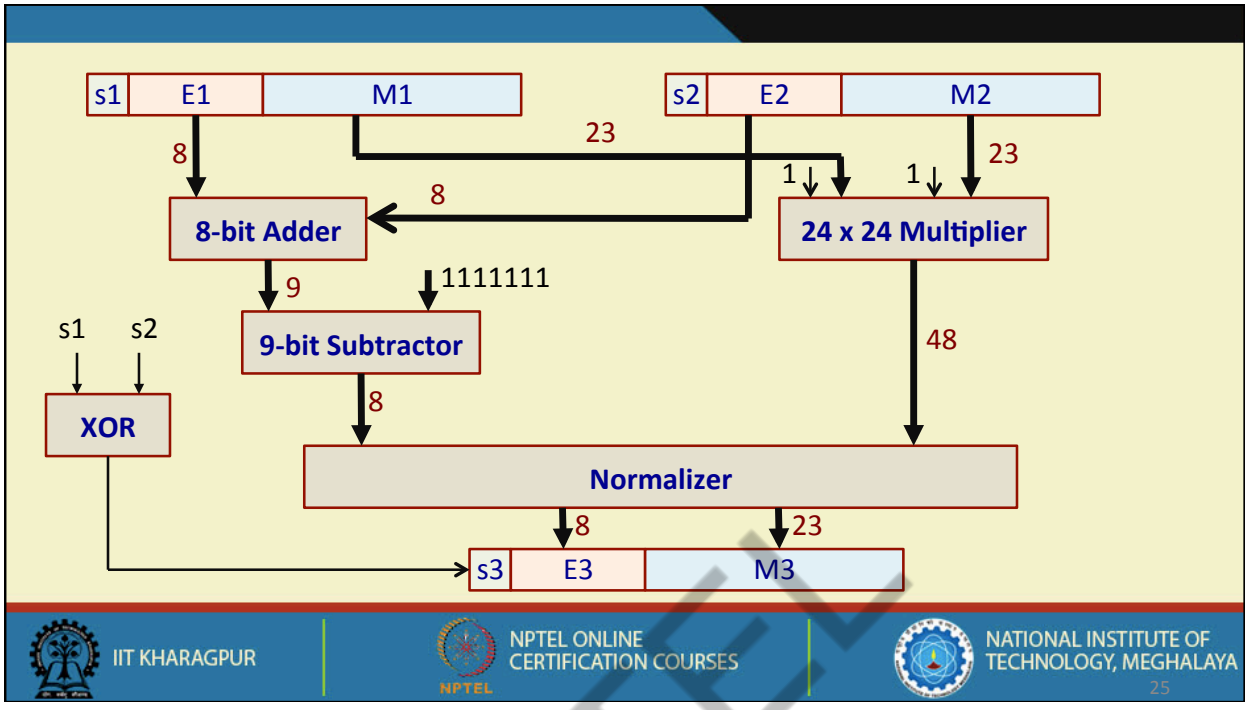
- Two numbers: $M1 \times 2^{E1}$ and $M2 \times 2^{E2}$
- Basic steps:
  - Add the exponents *E1* and *E2* and subtract the *BIAS*.
  - Multiply *M1* and *M2* and determine the sign of the result.
  - Normalize the resulting value, if necessary.

Multiplication Example

- Suppose we want to multiply F1 = 270.75 and F2 = -2.375

  $F1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$

  $F2 = (-2.375)_{10} = (-10.011)_2 = -1.0011 \times 2^1$
- Add the exponents: 8 + 1 = 9
- Multiply the mantissas: 1.01000001100001
- Result: $1.01000001100001 \times 2^9$



Floating-Point Division

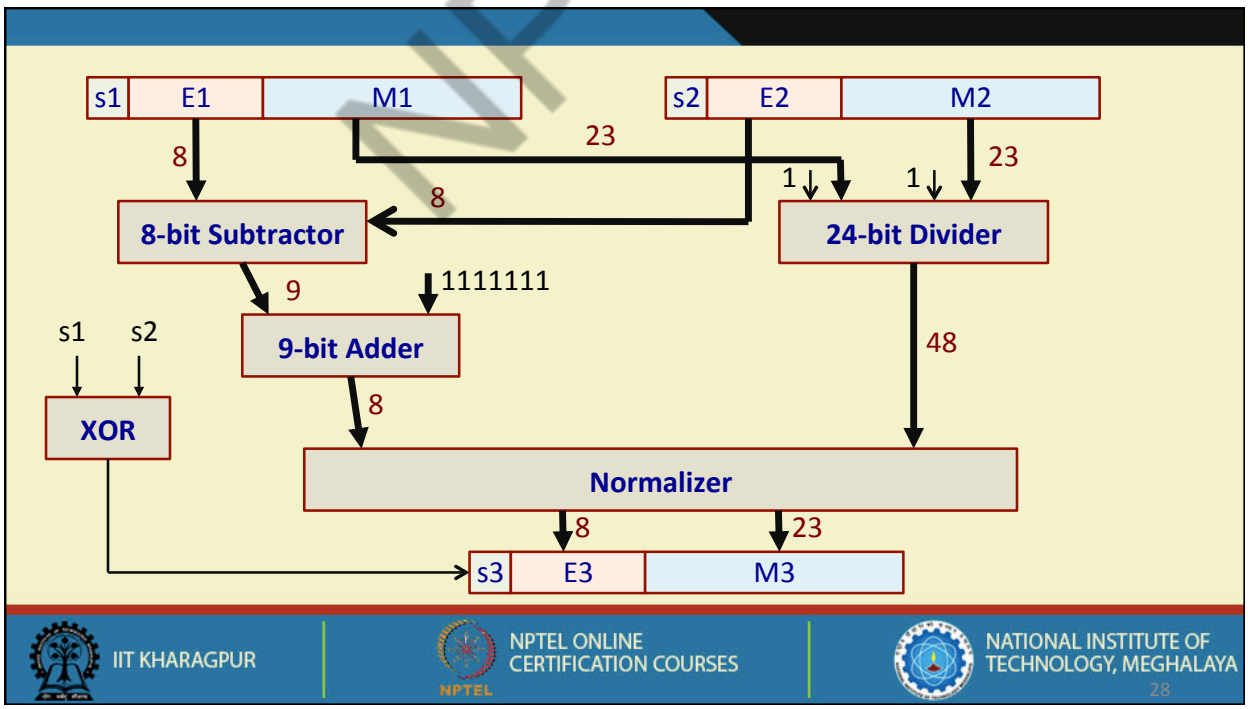
- Two numbers: $M1 \times 2^{E1}$ and $M2 \times 2^{E2}$
- Basic steps:
  - Subtract the exponents *E1* and *E2* and add the *BIAS*.
  - Divide *M1* by *M2* and determine the sign of the result.
  - Normalize the resulting value, if necessary.

Division Example

- Suppose we want to divide F1 = 270.75 by F2 = -2.375

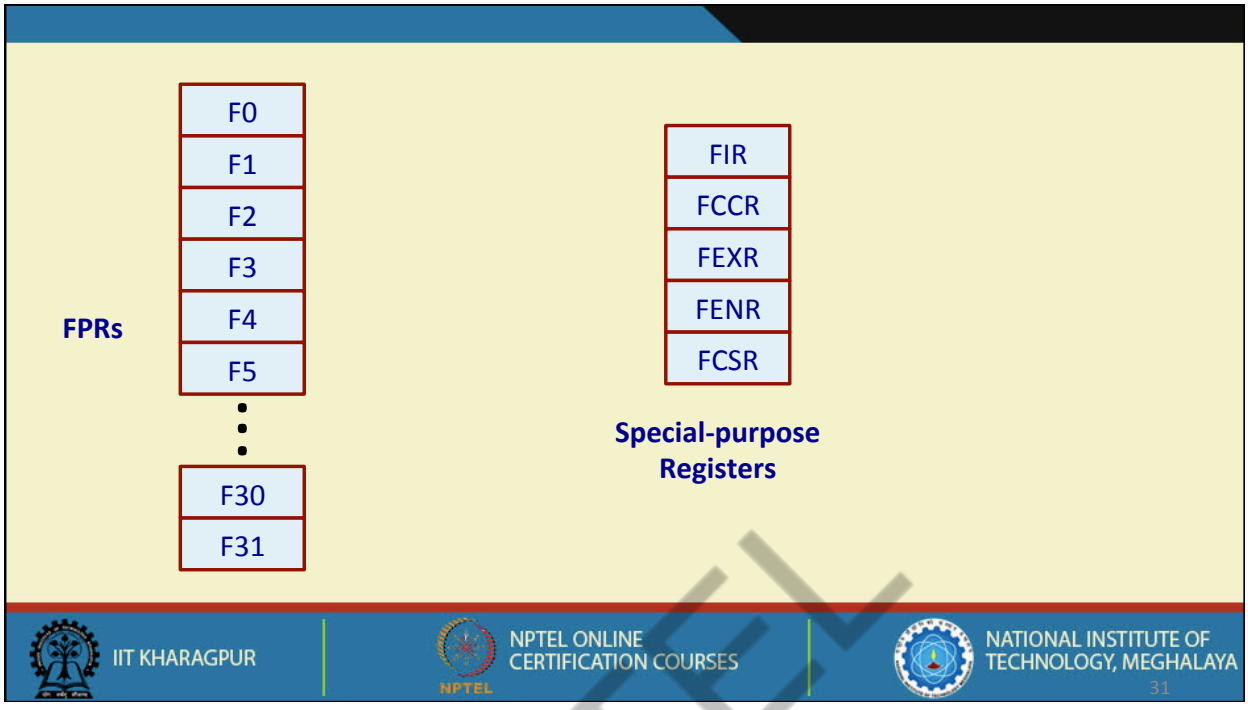
  $F1 = (270.75)_{10} = (100001110.11)_2 = 1.0000111011 \times 2^8$

  $F2 = (-2.375)_{10} = (-10.011)_2 = -1.0011 \times 2^1$
- Subtract the exponents: 8 – 1 = 7
- Divide the mantissas: 0.1110010
- Result: 0.1110010×2^7
- After normalization: 1.110010×2^6



FLOATING-POINT ARITHMETIC IN MIPS

- The MIPS32 architecture defines the following floating-point registers (FPRs).
  - 32 32-bit floating-point registers F0 to F31, each of which is capable of storing a single-precision floating-point number.
  - Double-precision floating-point numbers can be stored in even-odd pairs of FPRs (e.g., (F0,F1), (F10,F11), etc.).
- In addition, there are five special-purpose FPU control registers.



Typical Floating Point Instructions in MIPS

- Load and Store instructions
  - Load Word from memory
  - Load Double-word from memory
  - Store Word to memory
  - Store Double-word to memory
- Data Movement instructions
  - Move data between integer registers and floating-point registers
  - Move data between integer registers and floating-point control registers

IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA
32

- Arithmetic instructions
  - Floating-point absolute value
  - Floating-point compare
  - Floating-point negate
  - Floating-point add
  - Floating-point subtract
  - Floating-point multiply
  - Floating-point divide
  - Floating-point square root
  - Floating-point multiply add
  - Floating-point multiply subtract

- Rounding instructions:
  - Floating-point truncate
  - Floating-point ceiling
  - Floating-point floor
  - Floating-point round
- Format conversions:
  - Single-precision to double-precision
  - Double-precision to single-precision

Example: Add a scalar s to a vector A

```
for (i=1000; i>0; i--)
  A[i]= A[i] + s;
```

```
Loop:   L.D     F0, 0(R1)
        ADD.D   F4, F0, F2
        S.D     F4, 0(R1)
        ADDI    R1, R1, -8
        BNE     R1, R2, Loop
```

R1: initially points to A[1000]

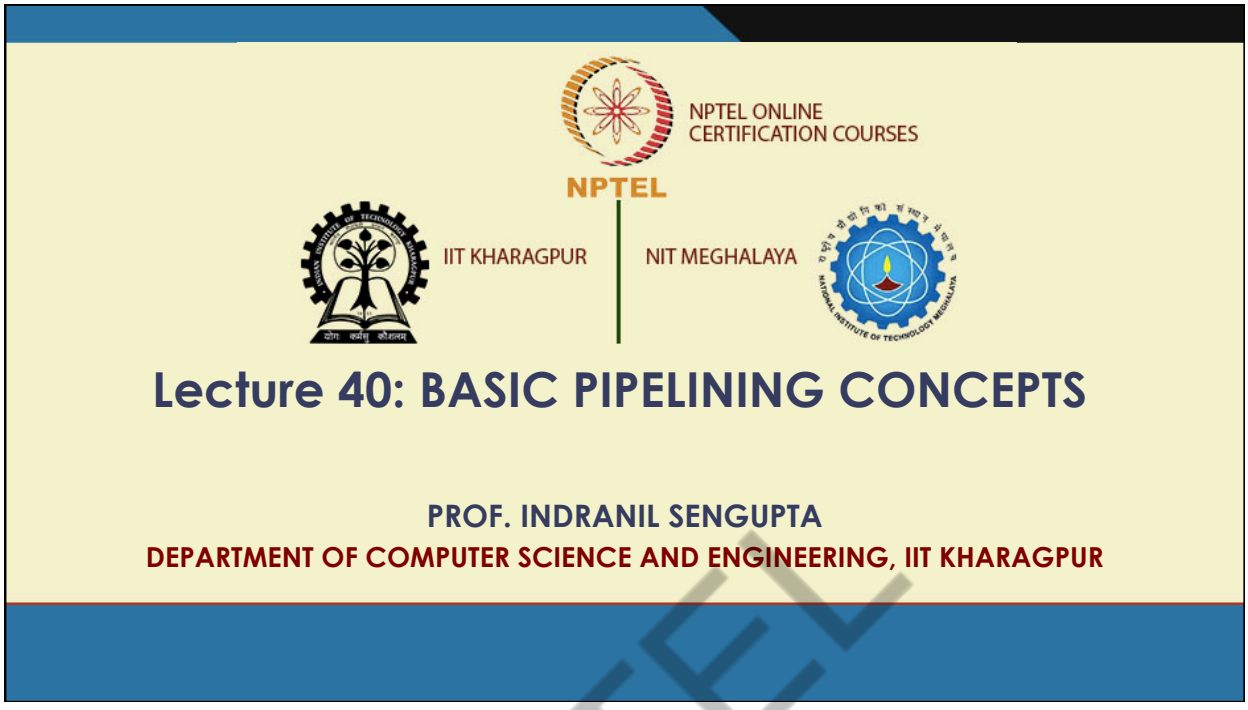
(F2,F3): contains the scalar s

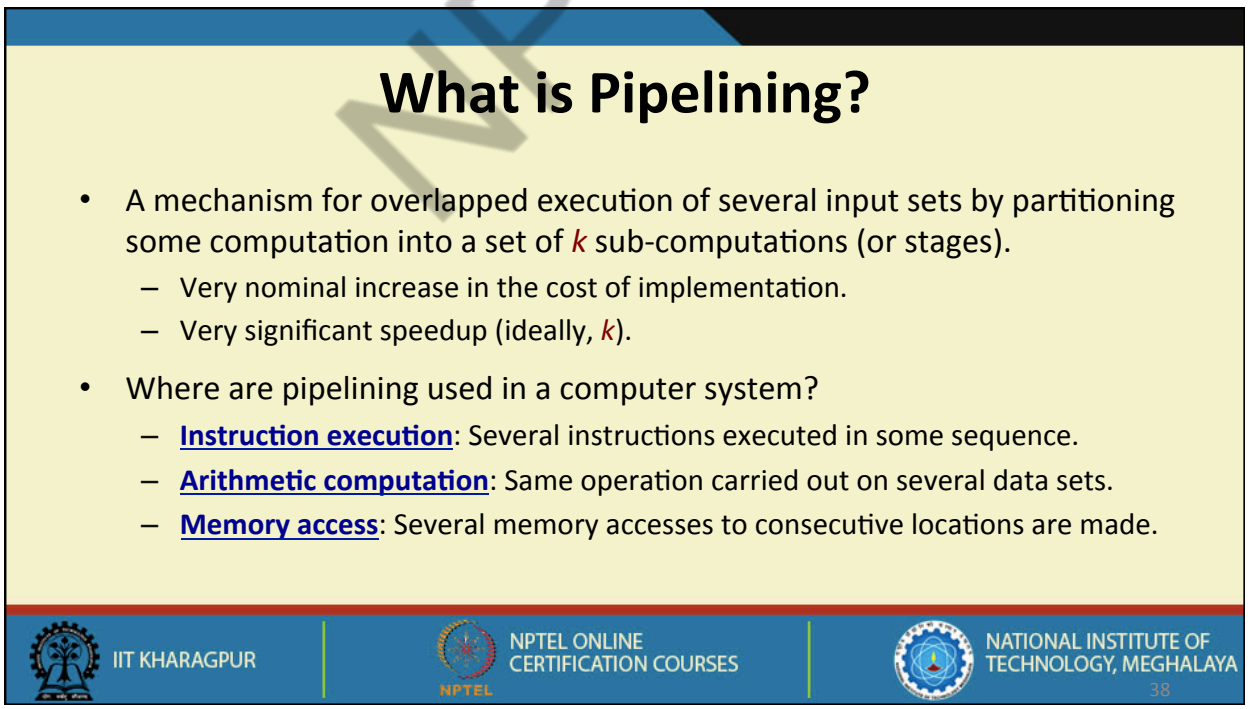
R2: initialized such that 8(R2) is the address of A[1]

We assume double precision (64 bits):

- Numbers stored in (F0,F1), (F2,F3), and (F4,F5).

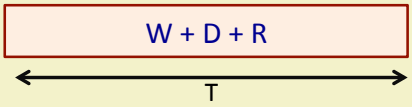
END OF LECTURE 39


NPTEL ONLINE
CERTIFICATION COURSES
NPTEL
IIT KHARAGPUR
NIT MEGHALAYA
Lecture 40: BASIC PIPELINING CONCEPTS
PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, IIT KHARAGPUR

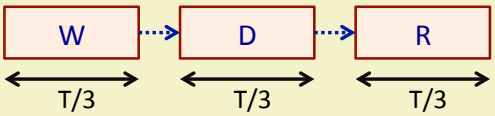

What is Pipelining?
• A mechanism for overlapped execution of several input sets by partitioning some computation into a set of k sub-computations (or stages).
– Very nominal increase in the cost of implementation.
– Very significant speedup (ideally, k).
• Where are pipelining used in a computer system?
– Instruction execution: Several instructions executed in some sequence.
– Arithmetic computation: Same operation carried out on several data sets.
– Memory access: Several memory accesses to consecutive locations are made.
IIT KHARAGPUR
NPTEL ONLINE CERTIFICATION COURSES
NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA
38

A Real-life Example

- Suppose you have built a machine *M* that can wash (*W*), dry (*D*), and iron (*R*) clothes, one cloth at a time.
  - Total time required is *T*.
- As an alternative, we split the machine into three smaller machines M_W, M_D and M_R, which can perform the specific task only.
  - Time required by each of the smaller machines is *T/3* (say).

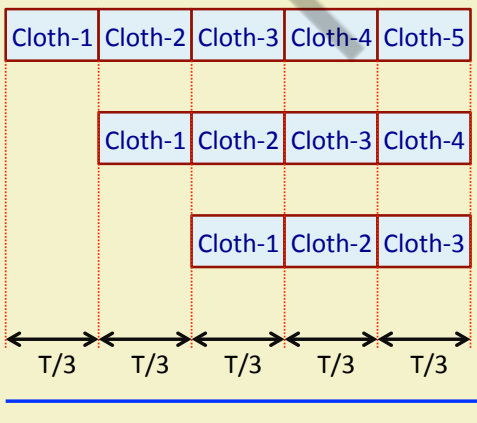


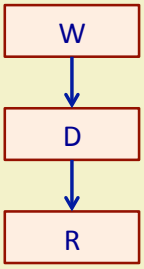
For N clothes, time T_1 = N.T



For N clothes, time T_3 = (2 + N).T/3

How does the pipeline work?





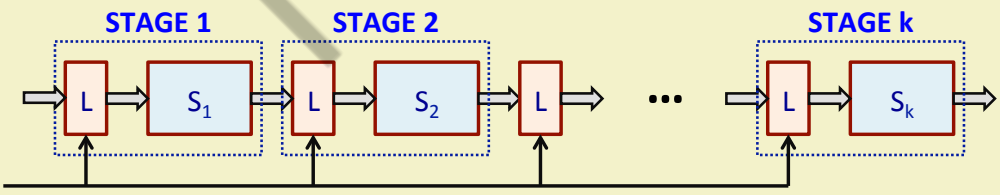
Finishing times:

- Cloth-1 – 3.T/3
- Cloth-2 – 4.T/3
- Cloth-3 – 5.T/3
- …
- Cloth-N – (2 + N).T/3

Extending the Concept to Processor Pipeline

- The same concept can be extended to hardware pipelines.
- Suppose we want to attain *k* times speedup for some computation.
  - **Alternative 1**: Replicate the hardware *k* times → cost also goes up *k* times.
  - **Alternative 2**: Split the computation into *k* stages → very nominal cost increase.
- Need for buffering:
  - In the washing example, we need a tray between machines (W & D, and D & R) to keep the cloth temporarily before it is accepted by the next machine.
  - Similarly in hardware pipeline, we need a *latch* between successive stages to hold the intermediate results temporarily.

Model of a Synchronous k-stage Pipeline



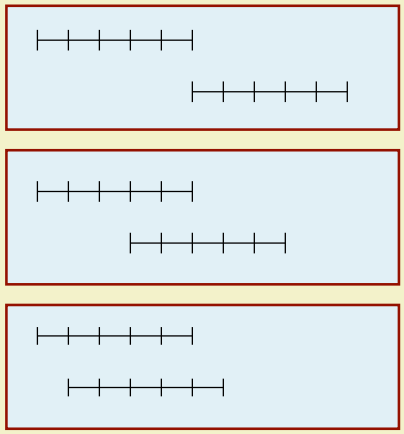
- The latches are made with master-slave flip-flops, and serve the purpose of isolating inputs from outputs.
- The pipeline stages are typically combinational circuits.
- When *Clock* is applied, all latches transfer data to the next stage simultaneously.

Types of Pipelined Processors

- Can be classified based on various parameters:
  a) Degree of overlap
     - Serial, overlapped or pipelined
  b) Depth of the pipeline
     - Shallow or Deep
  c) Structure of the pipeline
     - Linear or Non-linear
  d) How the operations are scheduled in the pipeline?
     - Static or Dynamic

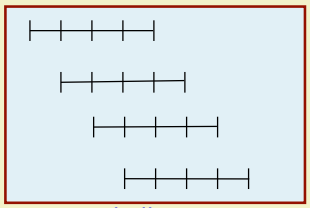
(a) Degree of Overlap

- Serial
  - The next operation can start only after the previous operation finishes.
- Overlapped
  - There is some overlap between successive operations.
- Pipelined
  - Fine-grain overlap between successive operations.

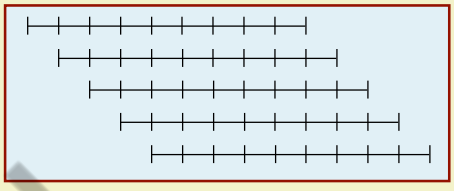


(b) Depth of the Pipeline

- Performance of a pipeline depends on the number of stages and how they can be utilized without conflict.
- Shallow pipeline is one with fewer number of stages.
  - Individual stages more complex.
- Deep pipeline is one with larger number of stages.
  - Individual stages simpler.



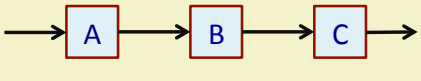
Shallow

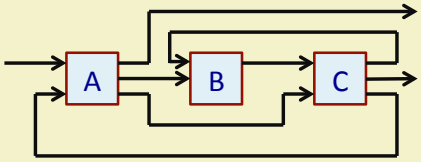


Deep

(c) Structure of the Pipeline

- **Linear Pipeline**: The stages that constitute the pipeline are executed one by one in sequence (say, from left to right).
- **Non-linear Pipeline**: The stages may not execute in a linear sequence (say, a stage may execute more than once for a given data set).





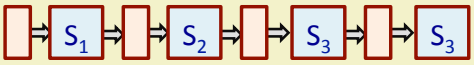
A possible sequence: A, B, C, B, C, A, C, A

(d) Scheduling Alternatives

- Static Pipeline:
  - Same sequence of pipeline stages are executed for all data / instructions.
  - If one data / instruction stalls, all subsequent ones also gets delayed.
- Dynamic Pipeline:
  - Can be reconfigured to perform variable functions at different times.
  - Allows feedforward and feedback connections between stages.

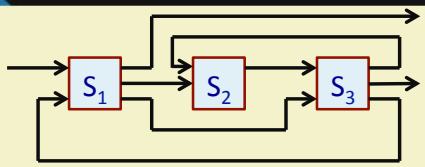
Reservation Table

- The *Reservation Table* is a data structure that represents the utilization pattern of successive stages in a synchronous pipeline.
  - Basically a space-time diagram of the pipeline that shows precedence relationships among pipeline stages.
    - X-axis shows the time steps
    - Y-axis shows the stages
  - Number of columns give evaluation time.
  - The reservation table for a 4-stage linear pipeline is shown.



	1	2	3	4
S_1	X			
S_2		X		
S_3			X	
S_4				X

- Reservation table for a 3-stage dynamic multi-function pipeline is shown.
  - Contains feedforward and feedback connections.
  - Two functions X and Y.
- Some characteristics:
  - *Multiple X's in a row* :: repeated use of the same stage in different cycles.
  - *Contiguous X's in a row* :: extended use of a stage over more than one cycles.
  - *Multiple X's in a column* :: multiple stages are used in parallel during a clock cycle.



	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X				
S_3			X		X		X	

	1	2	3	4	5	6
S_1	Y				Y	
S_2			Y			
S_3		Y		Y		Y

Speedup and Efficiency

Some notations:

τ :: clock period of the pipeline

t_i :: time delay of the circuitry in stage S_i

d_L :: delay of a latch

Maximum stage delay $\tau_m = \max \{t_i\}$

Thus, $\tau = \tau_m + d_L$

Pipeline frequency $f = 1 / \tau$

- If one result is expected to come out of the pipeline every clock cycle, *f* will represent the maximum throughput of the pipeline.

- The total time to process *N* data sets is given by

  $T_k = [(k-1) + N].\tau$ — $(k-1)\,\tau$ time required to fill the pipeline; 1 result every τ time after that → total $N.\tau$

- For an equivalent non-pipelined processor (i.e. one stage), the total time is

  $T_1 = N.k.\tau$ — (ignoring the latch overheads)

- Speedup of the k-stage pipeline over the equivalent non-pipelined processor:

$$S_k = \frac{T_1}{T_k} = \frac{N.k.\tau}{k.\tau + (N-1).\tau} = \frac{N.k}{k + (N-1)}$$

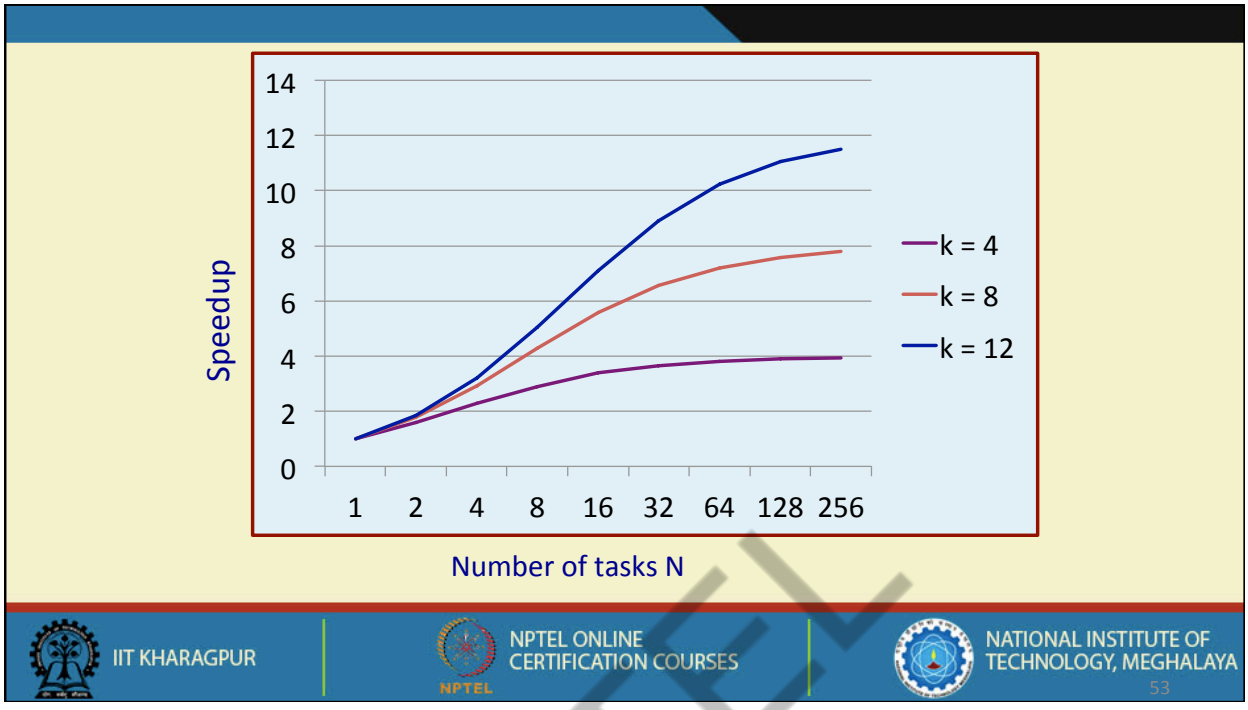
As $N \rightarrow \infty$, $S_k \rightarrow k$

- Pipeline efficiency:
  - How close is the performance to its ideal value?

$$E_k = \frac{S_k}{k} = \frac{N}{k + (N-1)}$$

- Pipeline throughput:
  - Number of operations completed per unit time.

$$H_k = \frac{N}{T_k} = \frac{N}{[k + (N-1)].\tau}$$



Clock Skew / Jitter / Setup time

- The minimum clock period of the pipeline must satisfy the inequality:

  $\tau \geq t_{skew+jitter} + t_{logic+setup}$

- Definitions:
  - *Skew*: Maximum delay difference between the arrival of clock signals at the stage latches.
  - *Jitter*: Maximum delay difference between the arrival of clock signal at the same latch.
  - *Logic delay*: Maximum delay of the slowest stage in the pipeline.
  - *Setup time*: Minimum time a signal needs to be stable at the input of a latch before it can be captured.

END OF LECTURE 40

NPTEL ONLINE CERTIFICATION COURSES

NATIONAL INSTITUTE OF TECHNOLOGY, MEGHALAYA

55

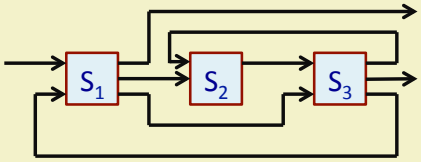
NPTEL ONLINE CERTIFICATION COURSES

IIT KHARAGPUR | NIT MEGHALAYA

Lecture 41: PIPELINE SCHEDULING

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, IIT KHARAGPUR

Scheduling of Non-linear Pipelines



	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X				
S_3			X		X		X	

Two operations X and Y

- **X: 8 time steps to complete**
- **Y: 6 time steps to complete**

	1	2	3	4	5	6
S_1	Y				Y	
S_2			Y			
S_3		Y		Y		Y

- Latency Analysis:
  - The number of time units between two initiations of a pipeline is called the *latency* between them.
  - Any attempt by two or more initiations to use the same pipeline stage at the same time will cause a *collision*.
  - The latencies that can cause collision are called *forbidden latencies*.
    - Distance between two X's in the same row of the reservation table.

	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X				
S_3			X		X		X	

Forbidden latencies: 2, 4, 5, 7

	1	2	3	4	5	6
S_1	Y				Y	
S_2						
S_3		Y		Y		Y

Forbidden latencies: 2, 4

– A *latency sequence* is a sequence of permissible non-forbidden latencies between successive task initiations.

– A *latency cycle* is a latency sequence that repeats the same subsequence.

Function X

- Forbidden latencies: 2, 4, 5, 7
- Possible latency cycles:
  - (1, 8) = 1, 8, 1, 8, … (average latency = 4.5)
  - (3) = 3, 3, 3, … (average latency = 3.0)
  - (6) = 6, 6, 6, … (average latency = 6.0)

Function Y

- Forbidden latencies: 2, 4
- Possible latency cycles:
  - (1, 5) = 1, 5, 1, 5, … (average latency = 3.0)
  - (3) = 3, 3, 3, … (average latency = 3.0)
  - (3, 5) = 3, 5, 3, 5, … (average latency = 4.0)

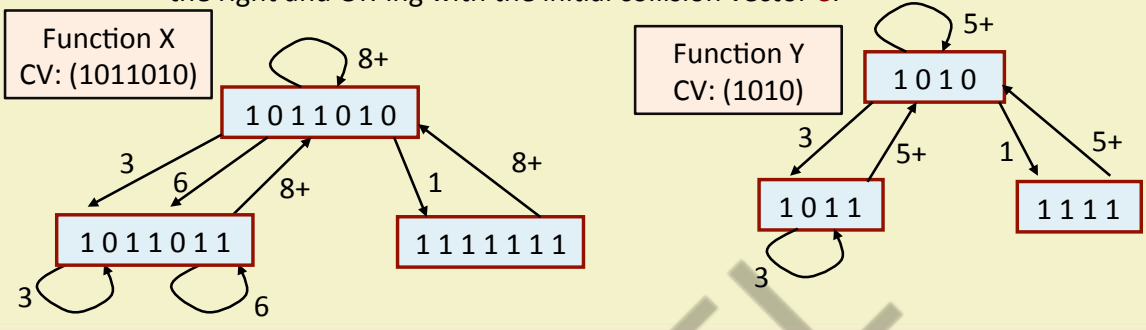
Constant Cycle

Collision Free Scheduling

- Main objective:
  – Obtain the shortest average latency between initiations without causing collisions.
- We define a *collision vector*.
  – If the reservation table has n columns, the maximum forbidden latency is $m \leq n-1$.
  – The permissible latencies p will satisfy: $1 \leq p \leq m-1$.
  – The collision vector is an m-bit binary vector $C = (C_m\ C_{m-1} \ldots C_2\ C_1)$, where $C_i = 1$ if latency i causes collision, and $C_i = 0$ otherwise.
  – C_m is always 1.

Function X: C_X = (1011010)
Function Y: C_Y = (1010)

- From the collision vector (CV), we can construct a *state diagram* specifying the permissible state transitions among successive initiations.
  - The collision vector corresponds to the initial state of the pipeline.
  - The next state at time *t+p* is obtained by shifting the present state *p*-bits to the right and OR-ing with the initial collision vector *C*.



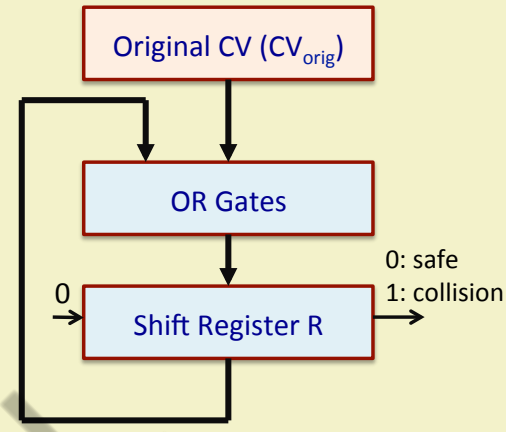
- From the state diagram, we can determine latency cycles that result in *minimum average latency (MAL)*.
  - In a *simple cycle*, a state appears only once.
  - Some of the simple cycles are *greedy cycles*, which are formed only using outgoing edges with minimum latencies.

Function X:	Function Y:
• Simple cycles: (3), (6), (8), (1,8), (3,8), (6,8)	• Simple cycles: (3), (5), (1,5), (3,5)
• Greedy cycles: (3), (1,8)	• Greedy cycles: (3), (1,5)

MAL = 3 for both X and Y

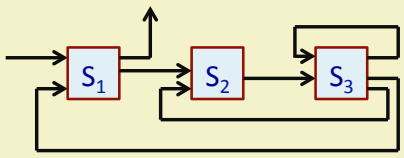
The Scheduling Algorithm

```
Load collision vector (CV) in a shift register R;
If (LSB of R is 1) then
    begin
        Do not initiate an operation;
        Shift R right by one position with 0 insertion;
    end
else
    begin
        Initiate an operation;
        Shift R right by one position with 0 insertion;
        R = R OR CV_orig;          // Logical OR with original CV
    end
```

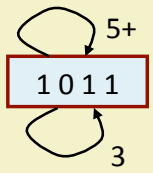


Optimizing a Pipeline Schedule

- We can insert non-compute (dummy) delay stages into the original pipeline.
  - This will modify the reservation table, resulting in a new collision vector.
  - Possibly a shorter MAL.

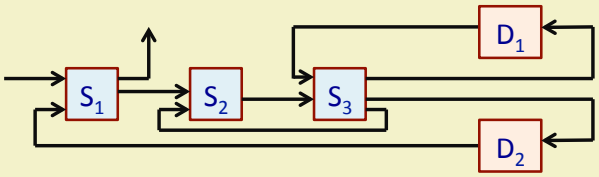


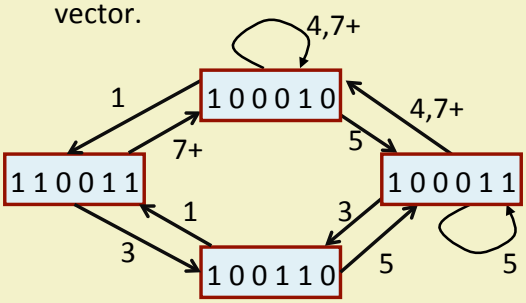
	1	2	3	4	5
S_1	X				X
S_2		X		X	
S_3			X	X	



MAL = 3

- Suppose we insert delay elements D_1 and D_2 as shown.
  - This will modify the reservation table, resulting in a new collision vector.





	1	2	3	4	5	6	7
S_1	X				→	→	X_2
S_2		X		X			
S_3			X	→	X_1		
D_1				X			
D_2						X	

MAL = 2 (for the greedy cycle (1,3))

Exercise 1

- For the following reservation tables,
  a) What are the forbidden latencies?
  b) Show the state transition diagram.
  c) List all the simple cycles and greedy cycles.
  d) Determine the optimal constant latency cycle, and the MAL.
  e) Determine the pipeline throughput, for τ = 20 ns.

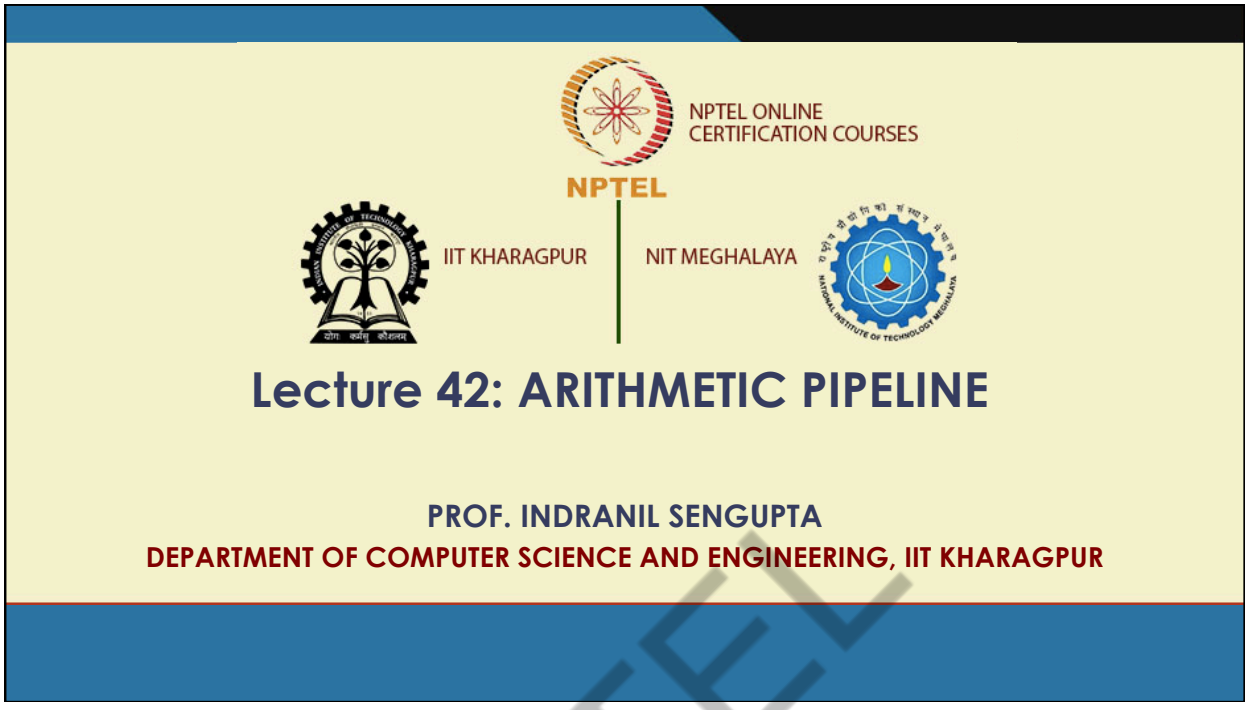
	1	2	3	4
S_1	X			X
S_2		X		
S_3			X	

	1	2	3	4	5	6	7
S_1	X		X				X
S_2		X			X		
S_3				X		X	

Exercise 2

- A non-pipelined processor X has a clock frequency of 250 MHz and an average CPI of 4. Processor Y, an improved version of X, is designed with a 5-stage linear instruction pipeline. However, due to latch delay and clock skew, the clock rate of Y is only 200 MHz.
  a) If a program consisting of 5000 instructions are executed on both processors, what will be the speedup of processor Y as compared to processor X?
  b) Calculate the MIPS rate of each processor during the execution of this particular program.

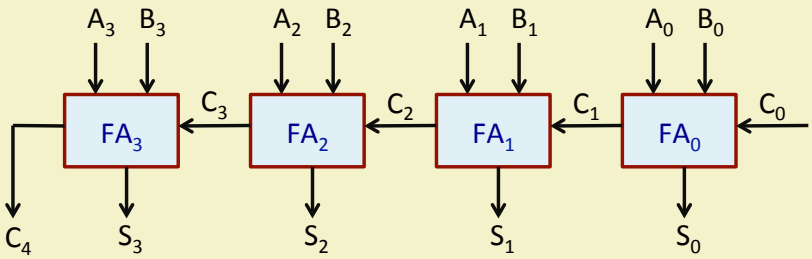
END OF LECTURE 41



Fixed Point Addition Pipeline

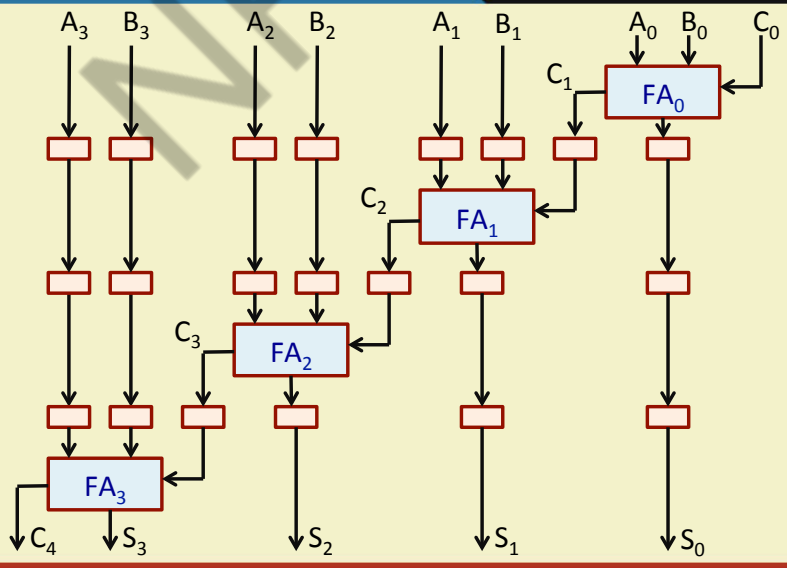
- We have seen how a ripple-carry adder works.
  - Rippling of the carries gives it a bad worst-case performance.
- We explore whether pipelining can improve the performance.
- *Assumption*: delay of a latch is comparable to the delay of a full adder.

A 4-bit Ripple Carry Adder



Worst-case delay ≈ 4 x (carry generation time in FA)

4-bit pipelined ripple-carry adder



- Delay of a full adder = t_{FA}
- Delay of a 1-bit latch = t_L
- Clock period $T \geq (t_{FA} + t_L)$
- After the pipeline is full, one result (sum) is generated every time T.
  - Convenient for vector addition kind of applications.

```
for (i=0; i<10000; i++)
    a[i] = b[i] + c[i];
```


Floating-Point Addition

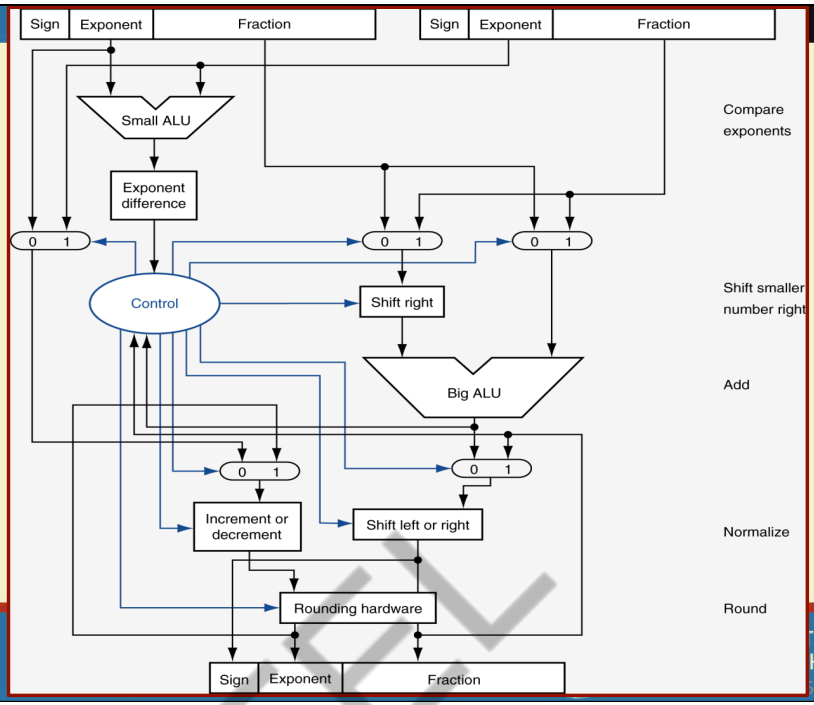
- Floating-point addition requires the following steps:
  a) Compare exponents and align mantissas.
  b) Add mantissas.
  c) Normalize result.
  d) Adjust exponent.
- Subtraction is similar.

Example:
A = 0.9504×10^3
B = 0.8200×10^2

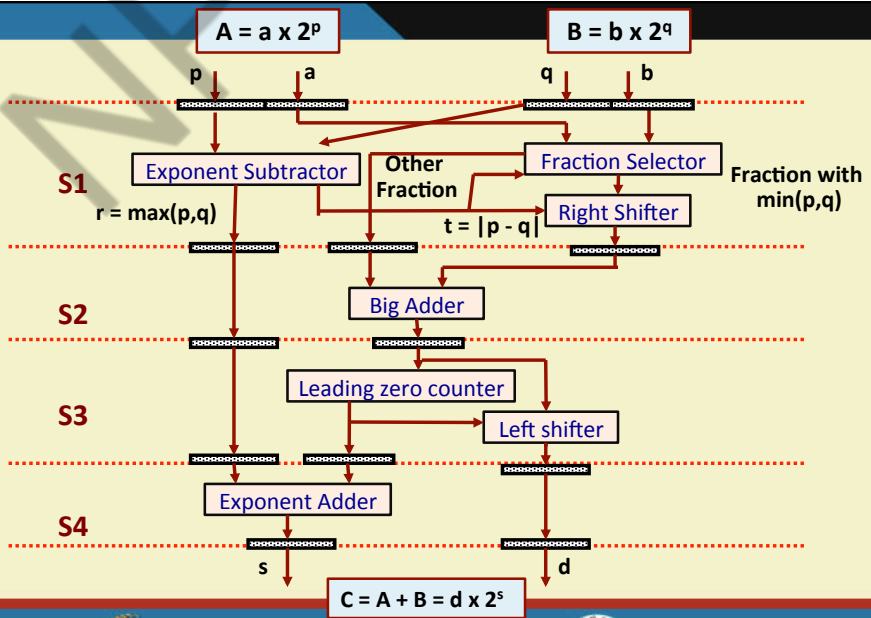
Align mantissa: 0.0820
Add mantissa: 0.9504 + 0.0820 = 1.0324
Normalize: 0.10324
Adjust exponent: 3 + 1 = 4
Sum = 0.10324×10^4

Floating-Point Addition Hardware

- The last step of rounding is required in IEEE-754 format.



4-Stage Floating Point Adder



Floating-Point Multiplication

- Floating-point multiplication requires the following steps:
  a) Add exponents.
  b) Multiply mantissas.
  c) Normalize result.
- Division is similar.

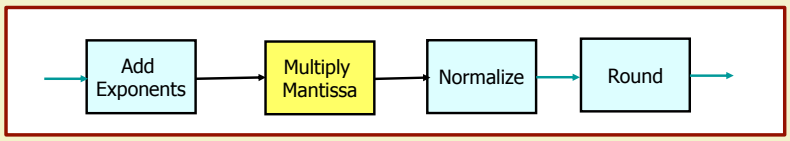
A last step of rounding is required in IEEE-754 format.

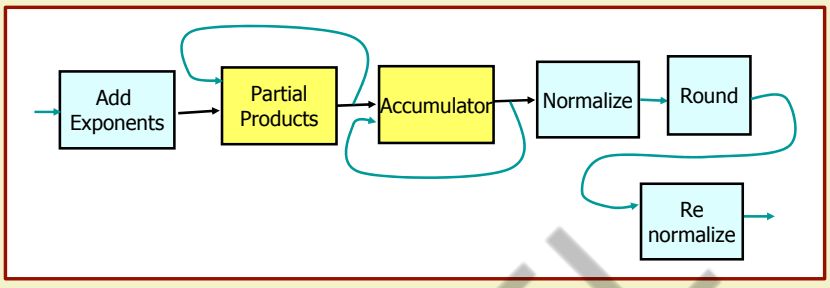
Example:
A = 0.9504×10^3
B = 0.8200×10^2

Add exponents: 3 + 2 = 5
Multiply mantissa: 0.9504 x 0.8200 = 0.7793
Normalize: 0.7793 (no change)
Product = 0.7793×10^5

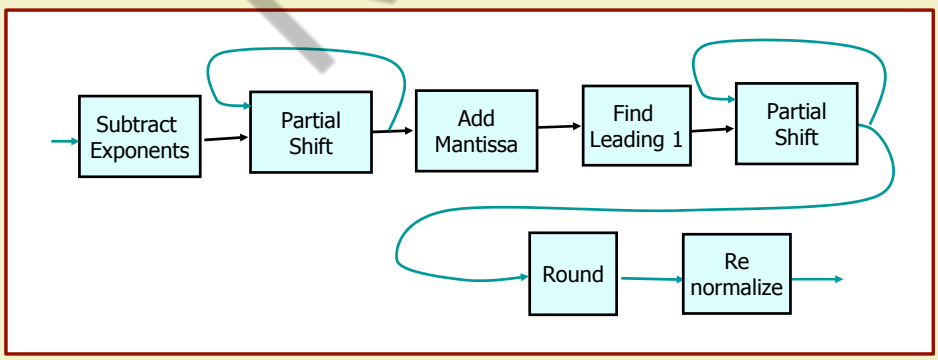
A MULTIFUNCTION PIPELINE FOR ADDITION AND MULTIPLICATION

Linear Pipeline for Floating-Point Multiply

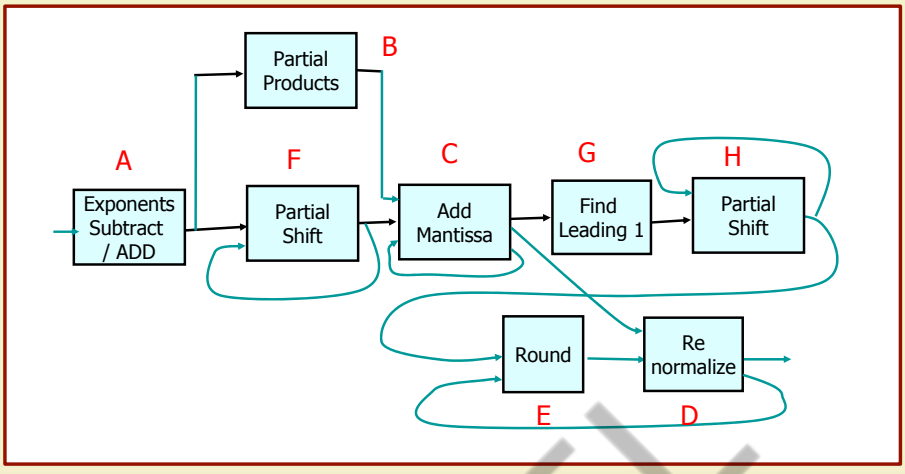




Linear Pipeline for Floating-Point Add



Combined Adder and Multiplier



Reservation Table for Multiply

	1	2	3	4	5	6	7
A	X						
B		X	X				
C			X	X			
D					X		X
E						X	
F							
G							
H							

- Forbidden latencies: 1, 2
- Collision Vector: (0 0 0 0 1 1)
- MAL = ?

Collision Scenarios

	1	2	3	4	5	6	7
A	X	Z					
B		X	X Z	Z			
C			X	X Z	Z		
D					X	Z	X
E						X	Z
F							
G							
H							

Latency 1 → collision

	1	2	3	4	5	6	7
A	X		Z				
B		X	X	Z	Z		
C			X	X	Z	Z	
D					X		X Z
E						X	
F							
G							
H							

Latency 2 → collision

	1	2	3	4	5	6	7
A	X			Z			
B		X	X		Z	Z	
C			X	X		Z	Z
D					X		X
E						X	
F							
G							
H							

Latency 3 → no collision

Reservation Table for Addition

	1	2	3	4	5	6	7	8	9
A	Y								
B									
C				Y					
D									Y
E								Y	
F		Y	Y						
G					Y				
H						Y	Y		

- Forbidden latencies: 1
- Collision Vector: (0 0 0 0 0 1)
- MAL = ?

Summary

- Arithmetic pipeline is a standard feature in modern-day processors.
- Mandatory for vector processors, which are designed specifically to operate on vectors of data.
- We shall see the impact of arithmetic pipeline in MIPS32 instruction pipeline implementation later.

END OF LECTURE 42