Lecture 17

EE 421 / CS 425

Digital System Design

Fall 2025

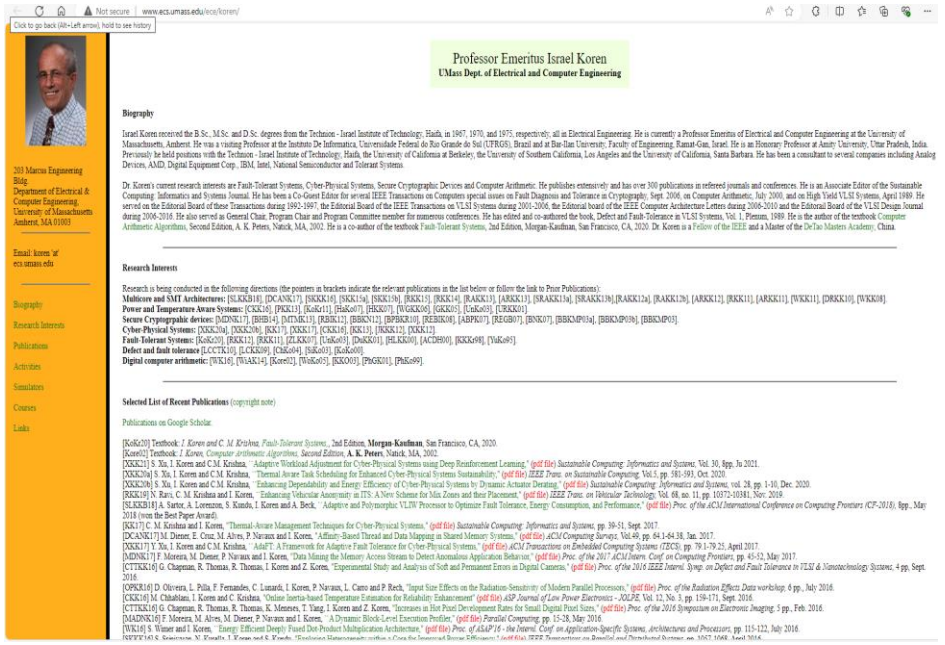
Shahid Masud

Topics

- Examples: Online tools for Binary Multiplication
- Binary Divider Operation
- Binary Divider Circuit
- STG of Divider
- Floating Point Representation (if time permits)
- Floating Point Multiplier – design and operation

Online Booth Encoding Simulator

http://www.ecs.umass.edu/ece/koren/arith/simulator/Booth/



Prof. Korean Page:

http://www.ecs.umass.edu/ece/koren/

Simulators Tab is on the left

Many simulators of Computer Arithmetic are available:

http://www.ecs.umass.edu/ece/koren/arith/simulator/

Online Modified Booth (Radix 4) Encoding

http://www.ecs.umass.edu/ece/koren/arith/simulator/ModBooth/

Simulator available on Prof Koren's website

Delay computation in binary array multiplier

Previous topic:

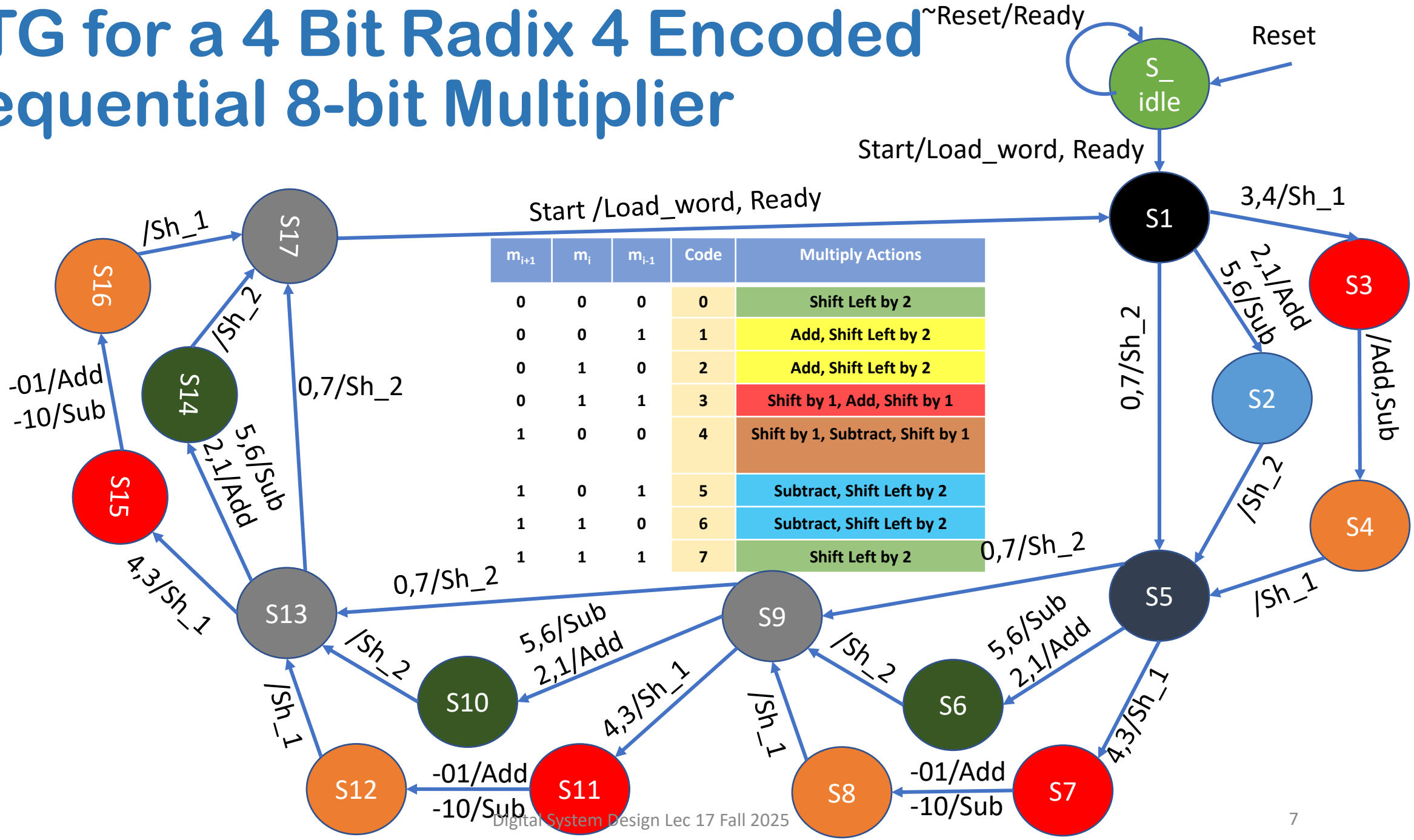
Delay computation in Array Multiplier (binary inputs):

http://www.ecs.umass.edu/ece/koren/arith/simulator/ArrMlt/

Radix 4 Coding for Multiplication

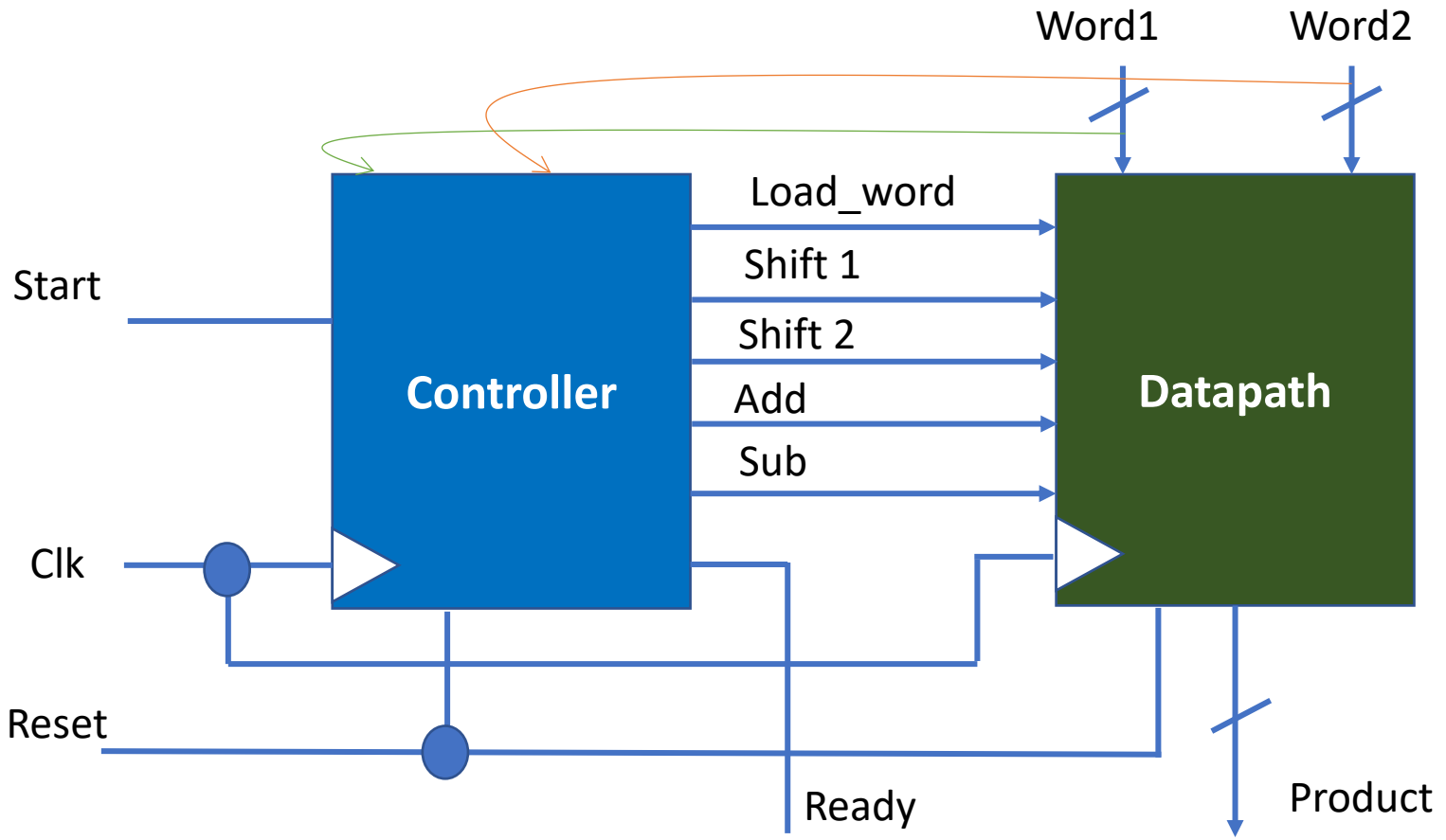
m_{i+1}	m_i	m_{i-1}	Code	Multiply Actions
0	0	0	0	Shift Left by 2
0	0	1	1	Add Multiplicand, Shift Left by 2
0	1	0	2	Add Multiplicand, Shift Left by 2
0	1	1	3	Shift by 1, Add Multiplicand, Shift by 1
1	0	0	4	Shift by 1, Subtract Multiplicand, Shift by 1
1	0	1	5	Subtract Multiplicand, Shift Left by 2
1	1	0	6	Subtract Multiplicand, Shift Left by 2
1	1	1	7	Shift Left by 2

STG for a 4 Bit Radix 4 Encoded Sequential 8-bit Multiplier



m_{i+1}	m_i	m_{i-1}	Code	Multiply Actions
0	0	0	0	Shift Left by 2
0	0	1	1	Add, Shift Left by 2
0	1	0	2	Add, Shift Left by 2
0	1	1	3	Shift by 1, Add, Shift by 1
1	0	0	4	Shift by 1, Subtract, Shift by 1
1	0	1	5	Subtract, Shift Left by 2
1	1	0	6	Subtract, Shift Left by 2
1	1	1	7	Shift Left by 2

Data Path Architecture of a Radix 4 Sequential Multiplier



Radix 4 Multiplication – Example 3

Show Radix 4 Encoded multiplication of 76 x 55, using 8 bits for both numbers

76 = 0100 1100
And 2's Compl is
-76= 1011 0100

55 = 0011 0111
And 2's Compl is
-55= 1100 1001

Convert 55 = 0011 0111to Radix 4 Encoded bits

Imagine Zero

55 = 0 0 1 1 0 1 1 1 [0]

RECODED
110 → 0-1
011 → 10
110 → 0-1
001 → 01

76 = Multiplicand
X 55 = Recoded Multiplier

										0	**1**	**0**	**0**	**1**	**1**	**0**	**0**
										0	**1**	**0**	**-1**	**1**	**0**	**0**	**-1**
	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1	0	0
	0	0	0	0	0	0	0	1	0	0	1	1	0	0	X	X	X
Partial Sum	0	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0
	1	1	1	1	1	1	0	1	1	0	1	0	0	X	X	X	X
Partial Sum	1	1	1	1	1	1	1	0	1	0	1	0	1	0	1	0	0
	0	0	0	0	1	0	0	1	1	0	0	X	X	X	X	X	X
Result	0	0	0	0	1	0	0	0	0	0	1	0	1	0	1	0	0

0 -1 = Sub, Shl2

1 0 = Shl1,Add, Shl1

0 -1 = Sub, Shl2

0 1 = Add, Shl2

Answer = 0001 0000 0101 0100 = (4+16+64+4096) = $(4180)_{10}$

Division Operation in Hardware

Division Operation in Decimal Numbers

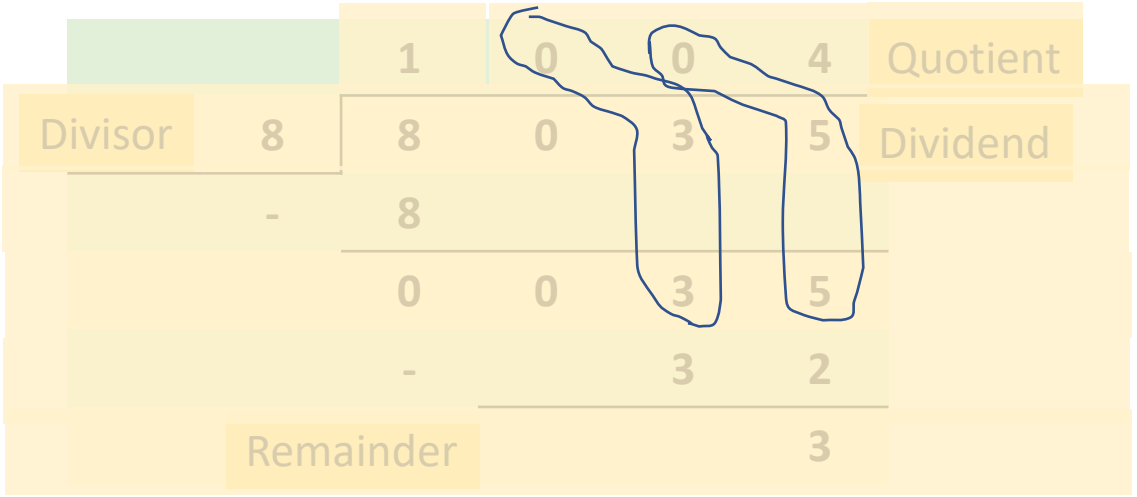
Division of 274 ÷ 13

				2	1	Quotient
Divisor	1	3	2	7	4	Dividend
		-	-2	6		
				1	4	
			-	1	3	
		Remainder			1	Rem

Division Operation in Decimal Numbers

Division of 299 ÷ 15

Decimal Division – another example



Division Operation in Binary – Example 1

Remainder

Division of 274 ÷ 13

			2	1
1	3	2	7	4
	-	-2	6	
			1	4
		-	1	3
				1 Rem

Quotient

Divisor

Dividend

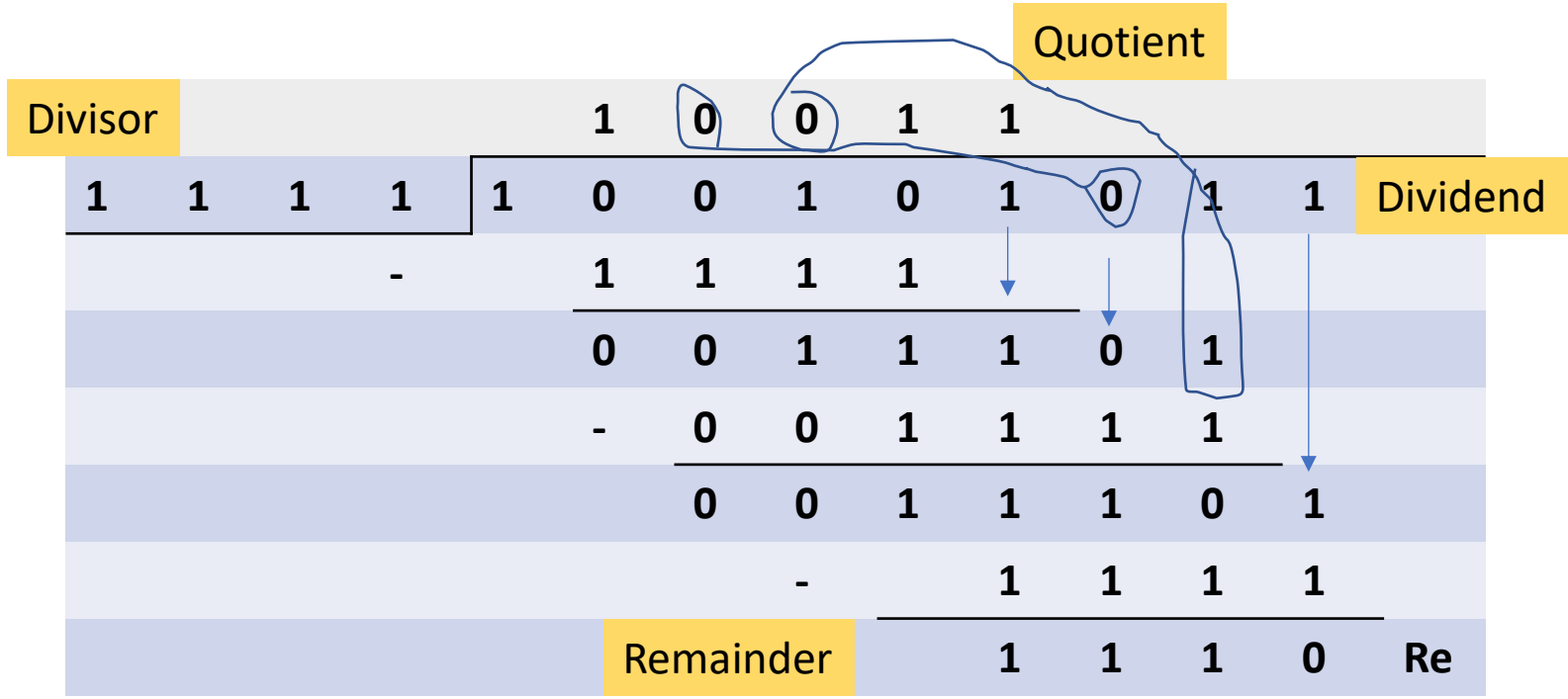
					1	0	1	0	1				
1	1	0	1	1	0	0	0	1	0	0	1	0	
			-	0	1	1	0	1					
						1	0	0	0	0			
					-		1	1	0	1			
						0	0	0	1	1	1	0	
							-		1	1	0	1	
									0	0	0	1	Re

Remainder

Division Operation in Binary – Example 2

Division of 299 ÷ 15

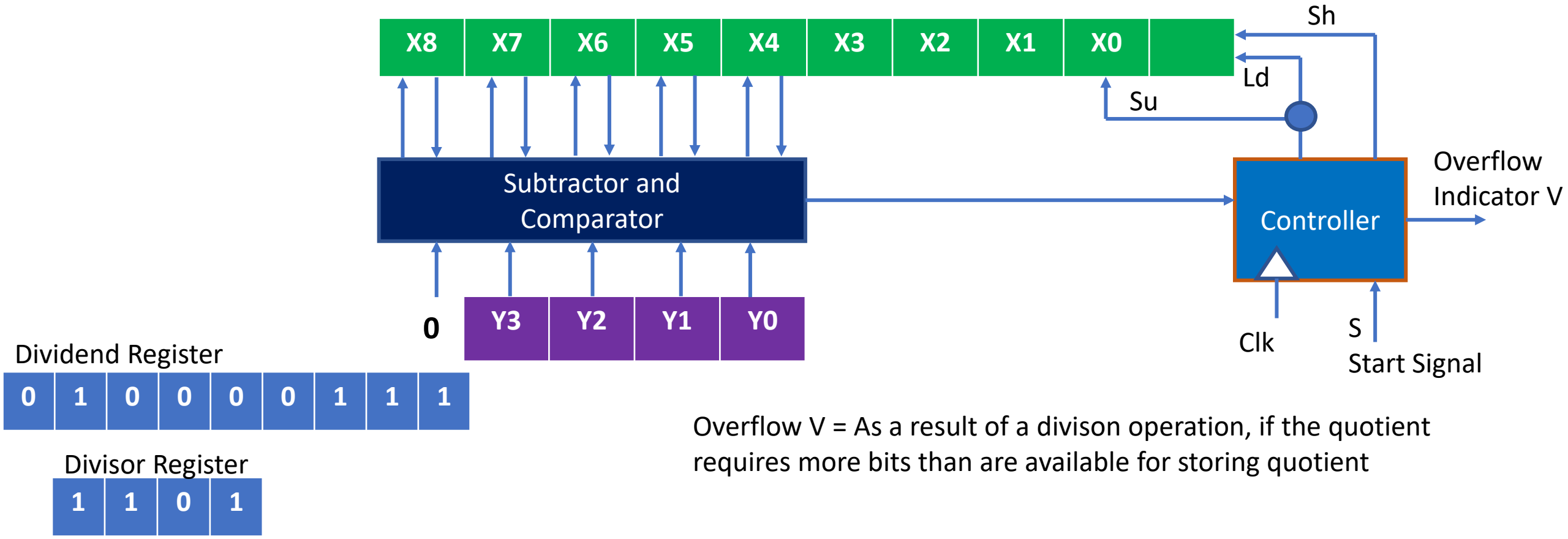
```
          1   9
    ------------
1   5 | 2   9   9
      -   2   8   5
    ------------
          1   4     Rem
```



Division Operation in Binary – Example 3

						1	1	0	1	Quotient			
Divisor	1	0	1	1	1	0	0	1	0	0	1	1	Dividend
				-		1	0	1	1				
						0	1	1	1	0			
				-			1	0	1	1			
							0	0	1	1	1	1	
					-				1	0	1	1	
					Remainder				0	1	0	0	Re

Block Diagram of Sequential Binary Divider



Overflow V = As a result of a divison operation, if the quotient requires more bits than are available for storing quotient

Operation of Sequential Binary Divider

Show binary division
135 ÷ 13

```
                1 0 1 0      <- Growing bits
1 1 0 1 | 1 0 0 0 0 1 1 1    <- Reducing bits
        -   1 1 0 1
        ---------
            0 1 1 1
        -   0 0 0 0
        ---------
            1 1 1 1
        -   1 1 0 1
        ---------
            0 1 0 1
        -   0 0 0 0
        ---------
            1 0 1
```

Remainder

Dividend

Dividing line between Dividend and Quotient

Divisor

After the shift, the right most position in dividend register is 'empty'

Subtraction is now carried out. The first quotient digit of 1 is stored in the unused portion of the dividend register

First quotient digit

Next we shift the dividend one place to the left

Since subtraction yields negative result so we shift dividend to the left again, and the second quotient bit remains 0

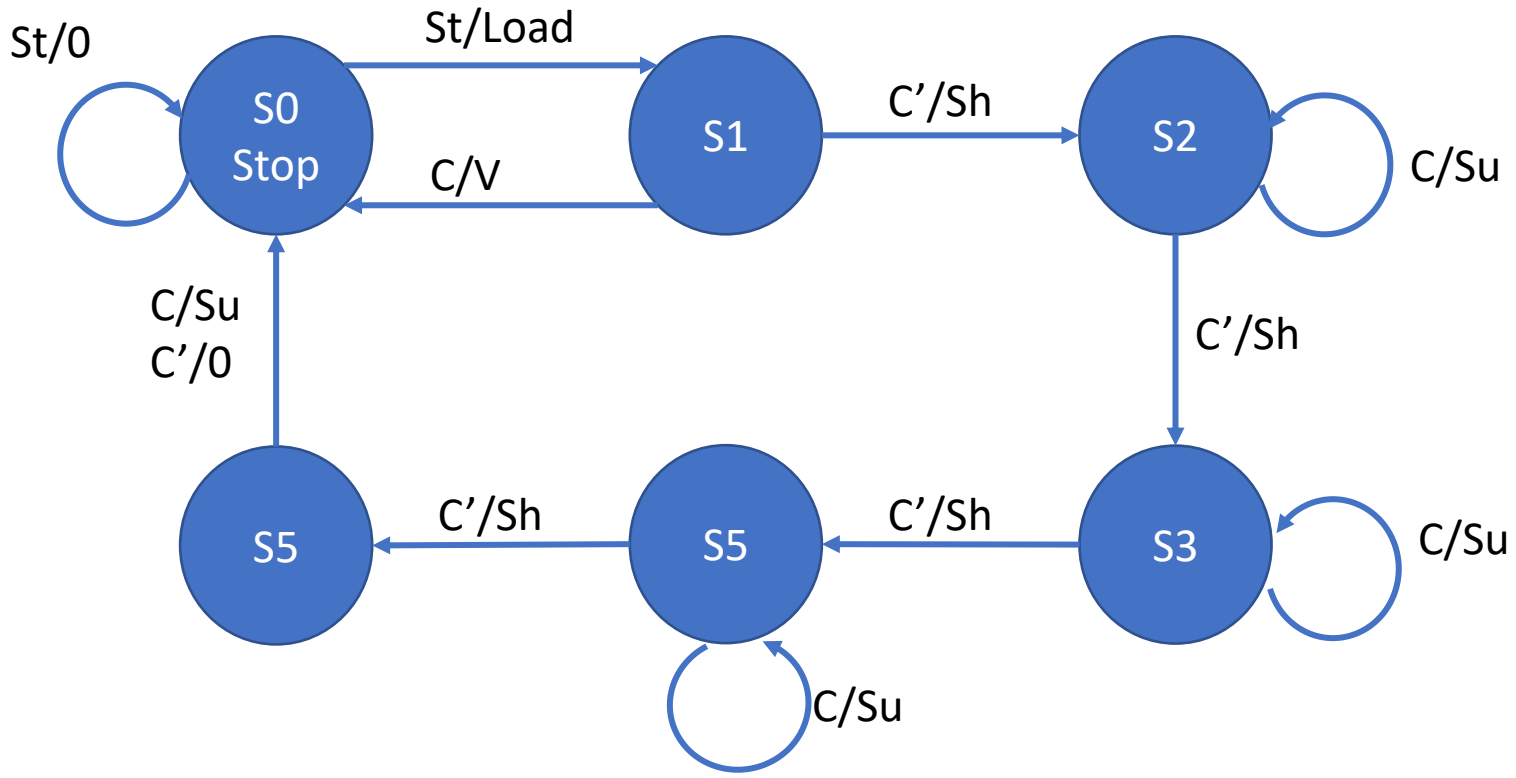
Subtraction is now carried out, the third quotient digit of 1 is stored in the unused portion of the dividend register

Third quotient digit

A final shift is done and fourth quotient bit is set to 0

Quotient

STG of a Binary Divider



Su = Subtract Signal
C = Comparator Output
If divisor is greater than 5 leftmost dividend bits (as per given number), then C=0; otherwise C=1
Whenever C=1, then subtract signal is generated and quotient bit is set to 1
Whenever C=0, then subtraction cannot occur without a negative result so a Shift signal Sh is generated

Division Examples

- Try using 2’s Complement Add instead of Sub in Division operations

Floating Point Operations in Hardware

Floating Point Arithmetic – Digital Design

$N=(-1)^S \times (1+F) \times 2^E$ E.g. $91.820734 \times 10^{-34}$

- A signed-magnitude system for the fractional part and a biased notation for the exponent
- Three subfields
  - Sign S
  - Fraction F (or Significand or Mantissa)
  - Exponent E
- Sign bit is 0 for positive numbers, 1 for negative numbers
- Fractions always start from **1**.xxxx, hence the integer **1** is not written (register has xxxx)
- Exponent is biased by +127 (add 127 to whatever is in register bits)
- Normalize: Express numbers is the standard format by shifting of bits and adding / subtracting from Exponent register

IEEE 754 Floating Point Representation

Single Precision IEEE 754

Sign	Exponent	Fraction	1 or 2 bits
1 bit	8 bits	23 bits	Guard Bits

32 bits

Double Precision IEEE 754

Sign	Exponent	Fraction	1 or 2 bits
1 bit	11 bits	52 bits	Guard Bits

64 bits

Examples of Floating Point Representation

$-(13.45)_{10}$
$= (1101.01\ 1100\ 1100\ 1100\ \ldots\ldots\ldots)^2$; this is un-normalized
$= (1.10101\ 1100\ 1100\ 1100\ 1100\ 1) \times 2^3$; normalized
Fraction part is 10101 1100 1100 1100 1
Biased Exponent is 3+127 = 130
Sign = 1

5.0345
= 101 . 0000 1000 1101 0100 1111 110; this is un-normalized
$= 1.\ 01\ 0000\ 1000\ 1101\ 0100\ 1111\ 110 \times 2^2$; normalized
Biased Exponent = 2+127 = 129 = $(1000\ 0001)_2$
Fraction = 01 0000 1000 1101 0100 1111 110
Sign = 0

Floating Point Multiplication

Consider two floating point numbers:
$(F_1 \times 2^{E1})$ and $(F_2 \times 2^{E2})$

The product of these two numbers is:

$= (F_1 \times 2^{E1}) \times (F_2 \times 2^{E2})$

$= (F_1 \times F_2) \times 2^{(E1+E2)}$

$= F \times 2^{E}$

Sign of result depends on Sign of the two numbers

Floating Point Multiplication Steps

1. **Normalize** the two numbers if not done already
2. The **exponents** of the **Multiplier** (E1) and the **multiplicand** (E2) bits are **added** and the base value is **subtracted** from the added result; The subtracted result is put in the exponential field of the result block → E1+E2-bias
3. **Multiply** the two fractions (or significands)
4. S1, the signed bit of the multiplicand is XOR'd with the multiplier signed bit of S2. The result is put into the resultant sign bit.
5. The **mantissa** of the Multiplier (M1) and multiplicand (M2) are **multiplied** and the result is placed in the resultant field of the mantissa (truncate/round the result for 24 bits) → M1 * M2
6. If the product is 0, adjust the proper representation of answer to 0
7. If the product fraction is too big, **Normalize** by shifting it right and incrementing the exponent
8. If the product fraction is too small, **Normalize** by shifting left and decrementing the exponent
9. Round to appropriate number of bits. If rounding results in loss of Normalization, then first **Normalize** and then do the rounding
10. If an exponent underflow (below -127) or overflow (above +127) occurs then generate an error condition

Bit-widths required

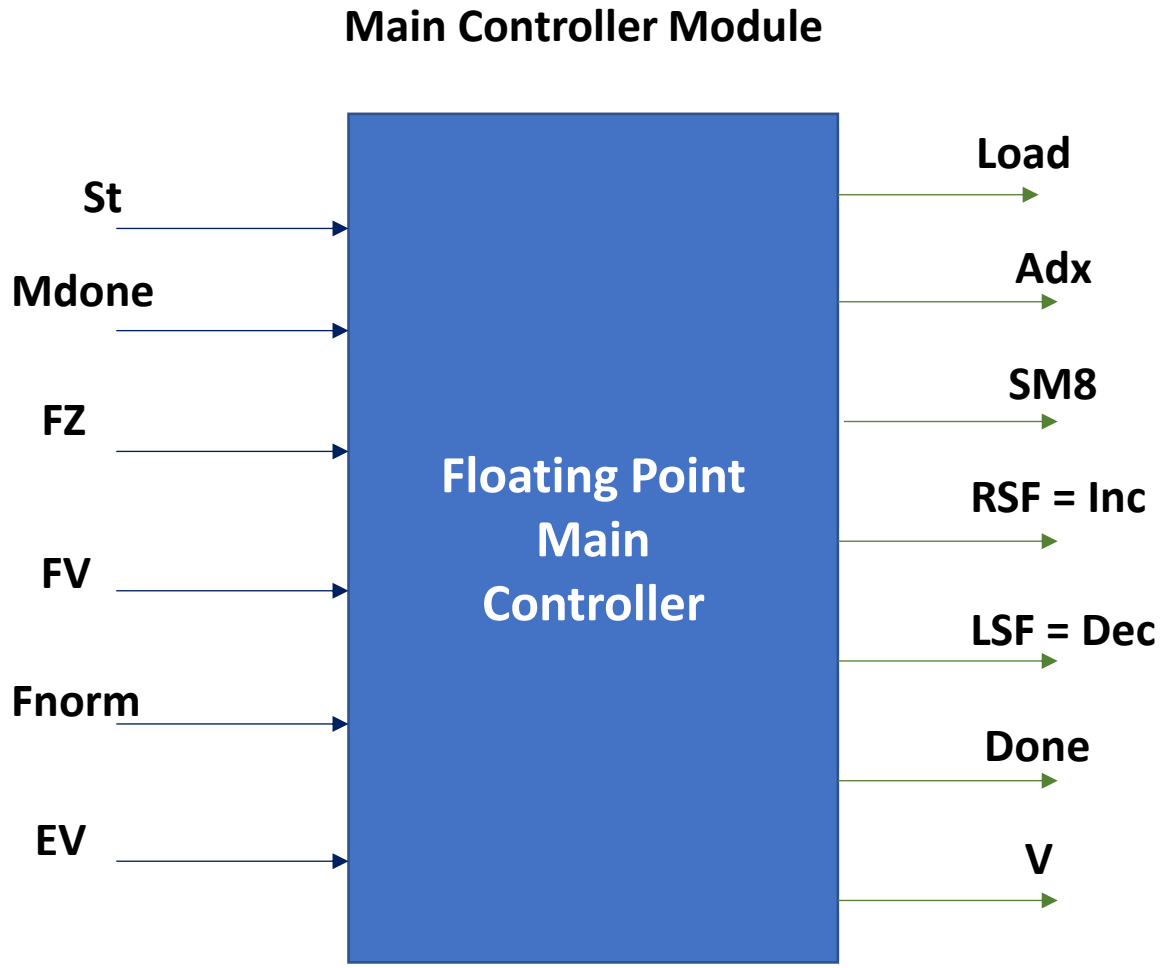
- Multiply the mantissa values including the hidden (**guard**) bits in rounding register. The resultant product of the 24 bits mantissas (M1 and M2) is 48 bits (2 bits are to the left of binary point)
- 8 bit adder (subtractor) needed for Exponent

Floating Point Multiplier Block Diagram - Example

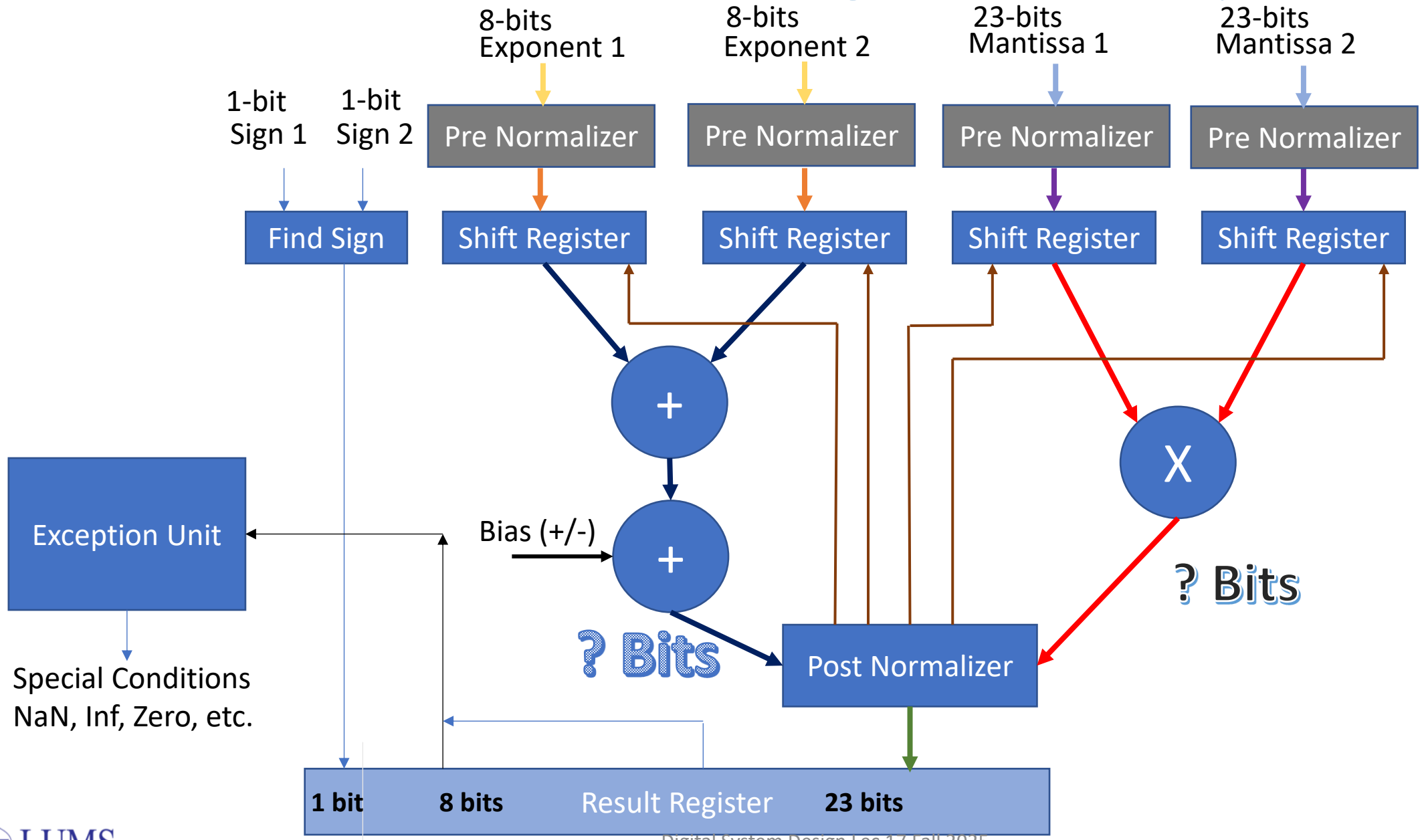
Definitions:

St: Start of Floating Point multiplication operation
Mdone: Fraction multiplication is complete
FZ: Fraction is Zero
FV: Fraction overflow (too small or too big)
Fnorm: F is normalized
EV: Exponent overflow

Load: Load F1, E1, F2, E2 into respective registers
Adx: Add Exponent; also start of multiplication
SM8: Set exponent to -8 to handle special case of 0
RSF: Shift fraction right; also increment E
LSF: Shift fraction left; also decrement E
V: overflow indicator
Done: Floating Point multiplication is complete



Data Path of Floating Point Multiplier



Floating Point with Booth Multiplier

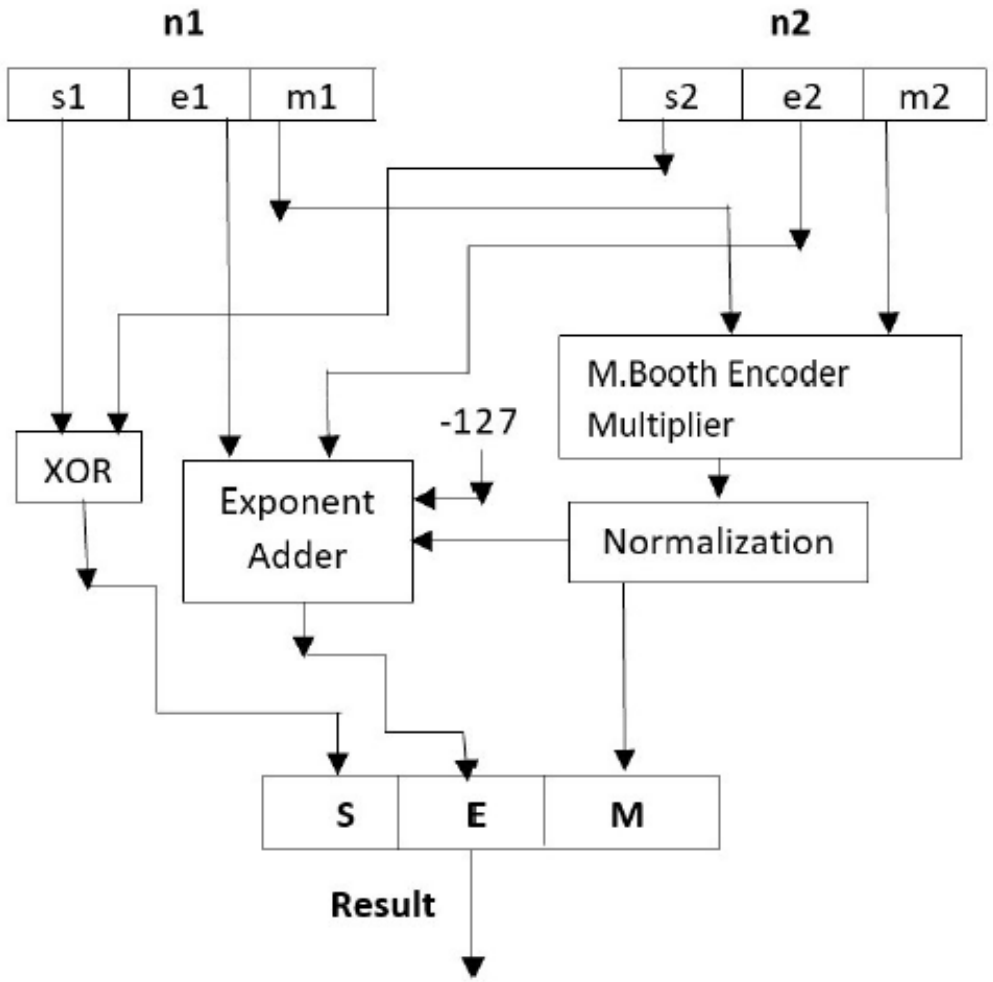
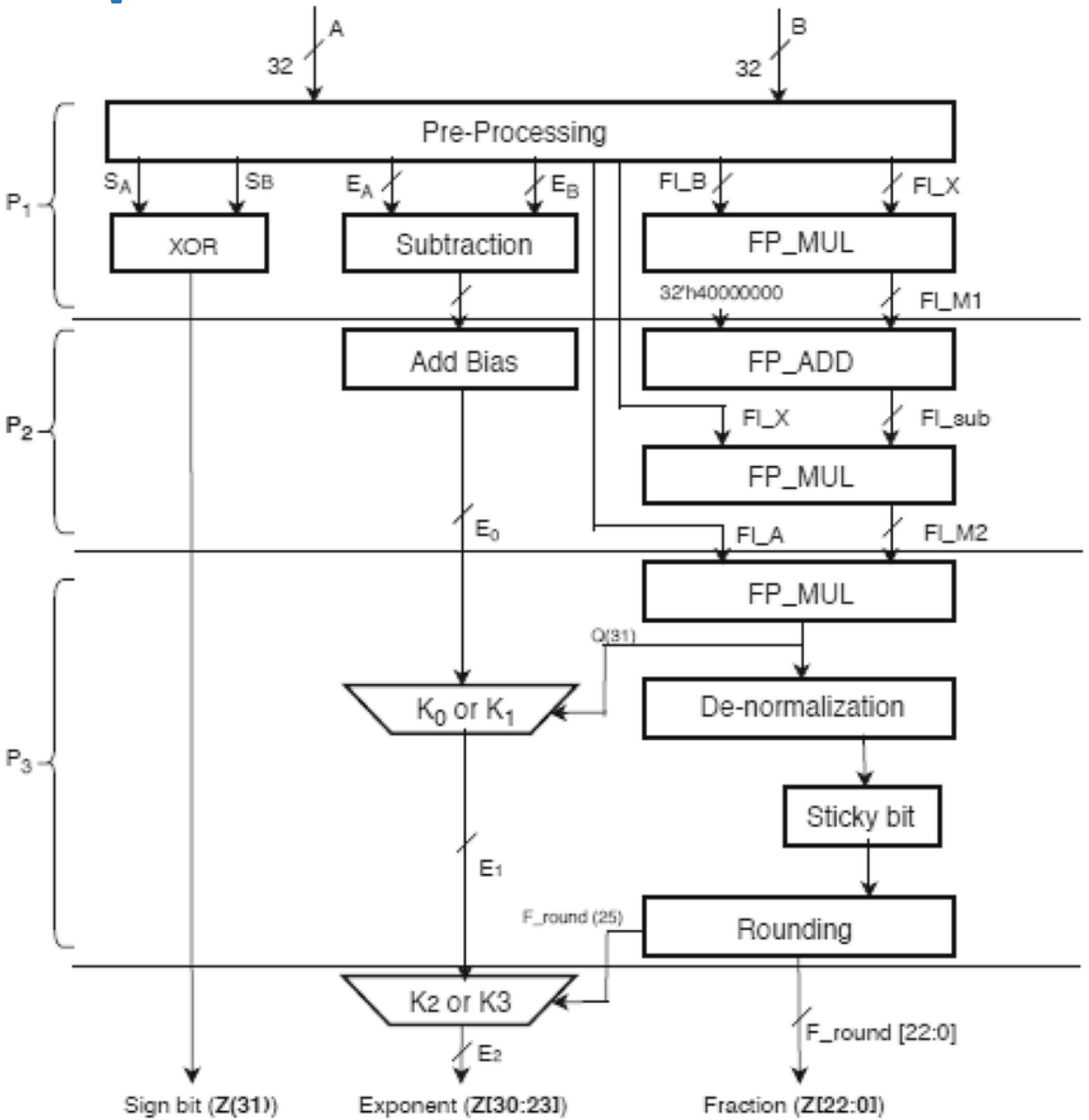
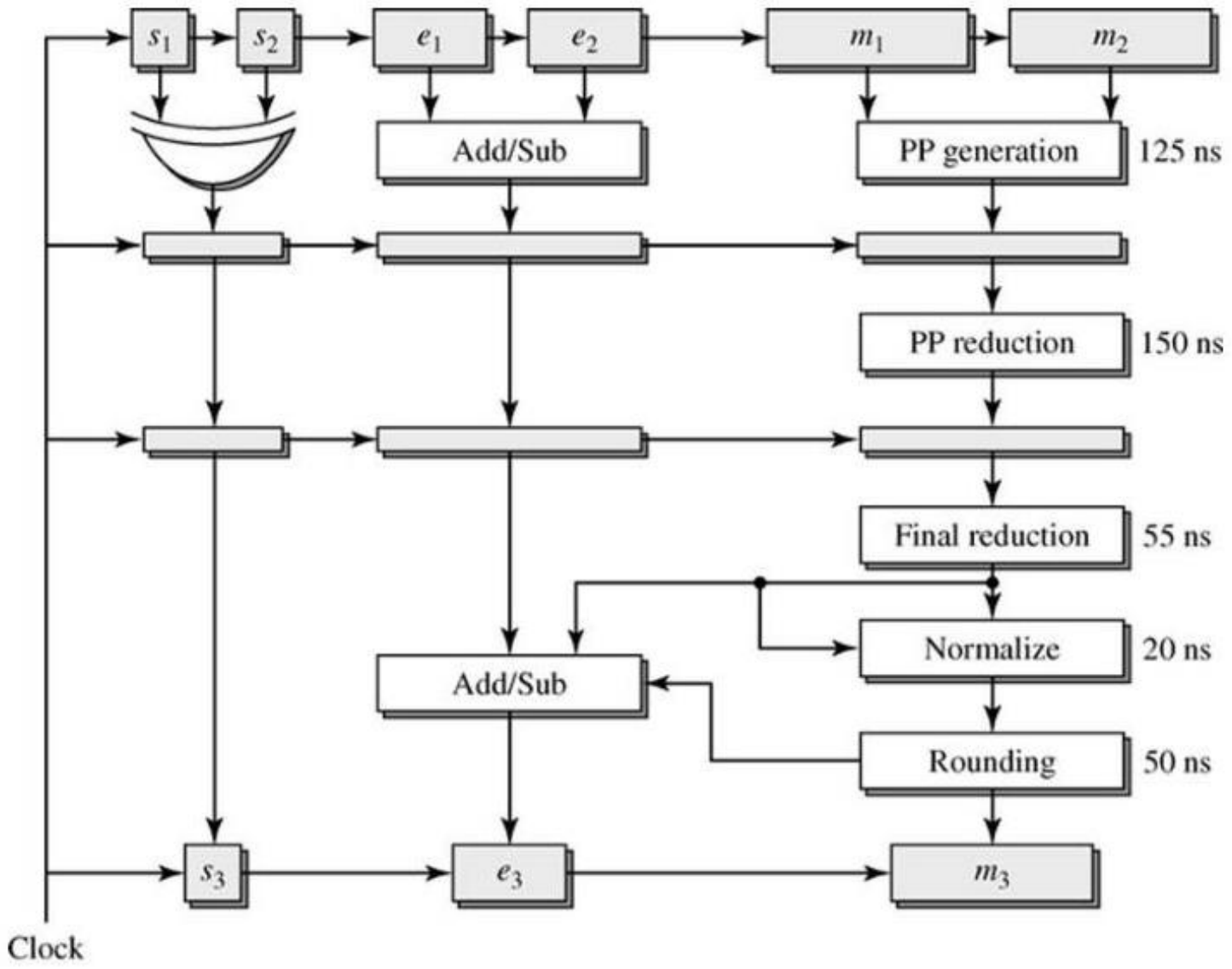


Fig. 4. Floating point Multiplication Block Diagram

Identifying Pipeline Cuts



Pipelined Floating Point Architecture



Operation Details of Datapath

Pre Normalize Block for Adder and Subtractor : Calculate the difference between the smaller and larger exponent. Adjust the smaller fraction by right shifting it, determine if the operation is an add or subtract after resolving the sign bits. Check for NaNs on inputs.

Pre Normalize Block for Multiplication: Computes the sum/difference of exponents, checks for exponent overflow, underflow condition and INF value on an input.

Post Normalize and Round Unit - Normalize fraction and exponent. Also do all the roundings in parallel and then pick the output corresponding to the chosen rounding mode.

Exceptions Unit – This unit Generates the exception signals like sNAN, qNAN, Inf and Ine The IEEE standard defines two classes of NaNs(non numbers): i. quiet NaNs (qNaNs) : A qNaN is a NaN with the most significant fraction bit set. ii. signaling NaNs (sNaNs): A sNaN is a NaN with the most significant fraction bit clear.

The single precision floating point format is divided into three main parts corresponding to the sign , exponent and mantissa.

Mutiplication of the two operands is done in three parts and thereby obtaining the Product.

The first part of the product which is the sign is determined by an exclusive OR function of the two input signs.

The exponent of the product which is the second part is calculated by adding the two input exponents. The third part which is the significand of the product is determined by multiplying the two input significands each with a '1'concatenated to it.

A multiplication of two floating-point numbers is done in four steps: • non-signed multiplication of mantissas: it must take account of the integer part, implicit in normalization. The number of bits of the result is twice the size of the operands (48 bits). • normalization of the result, the exponent can be modified accordingly . • addition of the exponents, taking into account the bias. • calculation of the sign.