DDCO

UNIT-4

CLASS NOTES

feedback/corrections: vibha@pesu.pes.edu

Vibha Masti

BINARY MULTIPLICATION

- In regular pen-paper multiplication

$$\begin{array}{rrr}
 & 1 & 3 \\
\times & 1 & 1 \\
\hline
 & 1 & 3 \\
1 & 3 & 0 \\
\hline
1 & 4 & 3
\end{array}$$

$$\begin{array}{lcccccccccl}
\text{4-bit} & & & & & 1 & 1 & 0 & 1 & & (13) \\
\text{4-bit} & & & & & 1 & 0 & 1 & 1 & & (11) \\
\cline{6-9}
 & & & & 1 & 1 & 1 & 0 & 1 & & \\
 & & & 1 & 1 & 1 & 0 & 1 & 0 & & \\
 & & 1 & 0 & 0 & 0 & 0 & 0 & 0 & & \\
 & & 1 & 1 & 0 & 1 & 0 & 0 & 0 & & \\
\cline{2-9}
 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & & (143) \\
\text{8-bit} & & & & & & & & & &
\end{array}$$

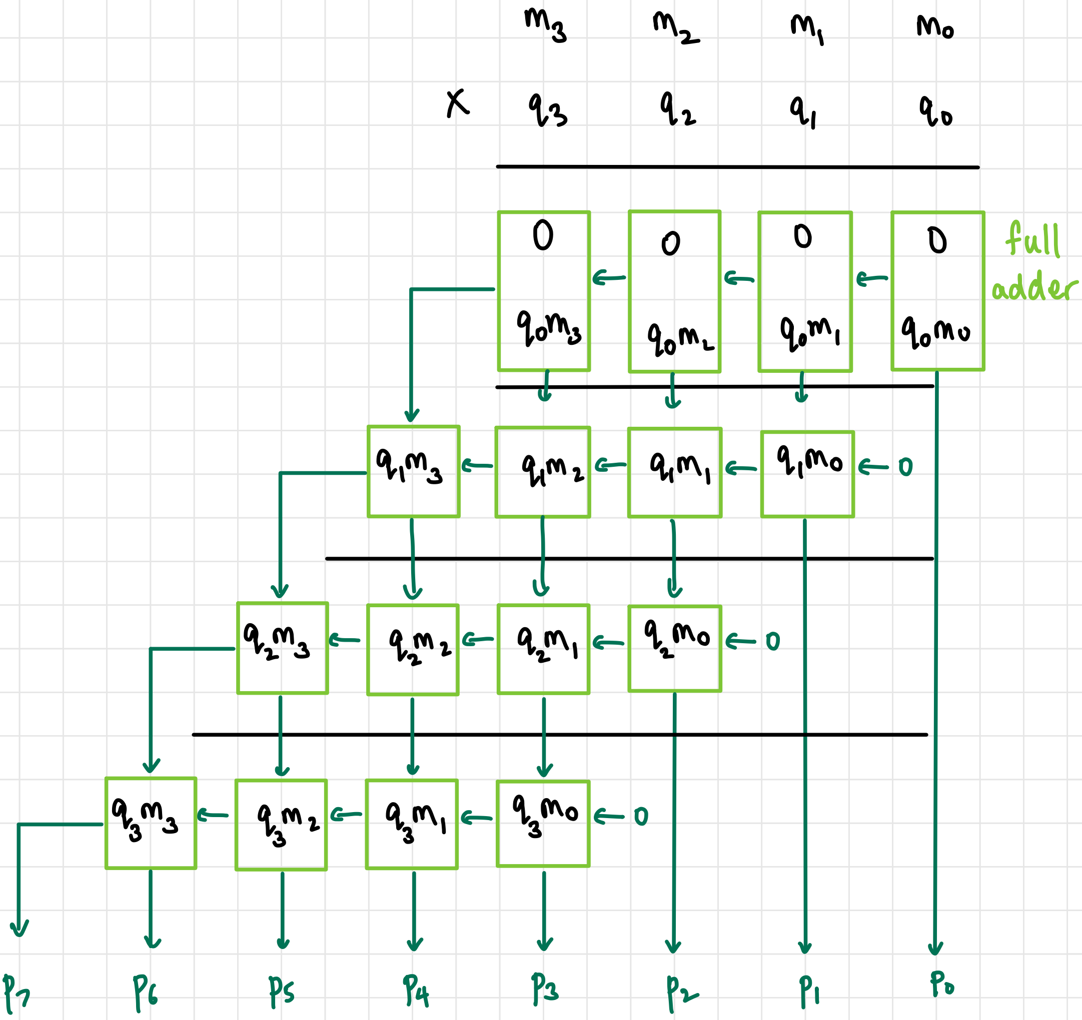
- The above algorithm works for unsigned numbers
- Product of two n-digit numbers ⟶ 2n digit result

- Multiplication of 2 binary numbers ⟶ AND function
- Product obtained by adding partial products
- Using array of combinational elements

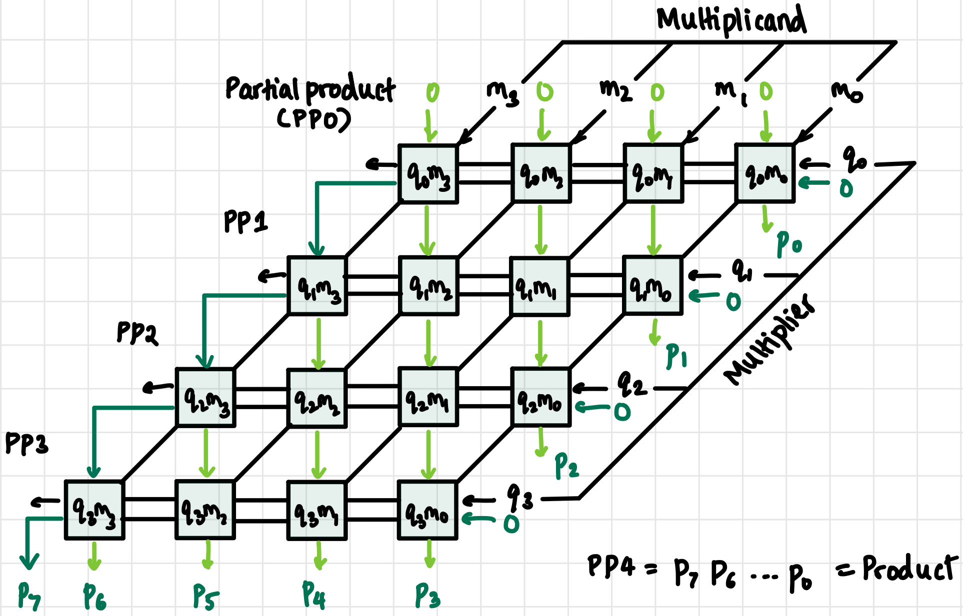
0 AND 0 ⟶ 0 × 0 ⟶ 0
0 AND 1 ⟶ 0 × 1 ⟶ 0
1 AND 0 ⟶ 1 × 0 ⟶ 0
1 AND 1 ⟶ 1 × 1 ⟶ 1

ARRAY MULTIPLICATION

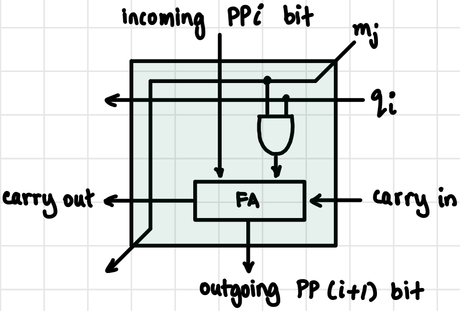
- Take initial product as 0000
- M— multiplicand, Q — multiplier



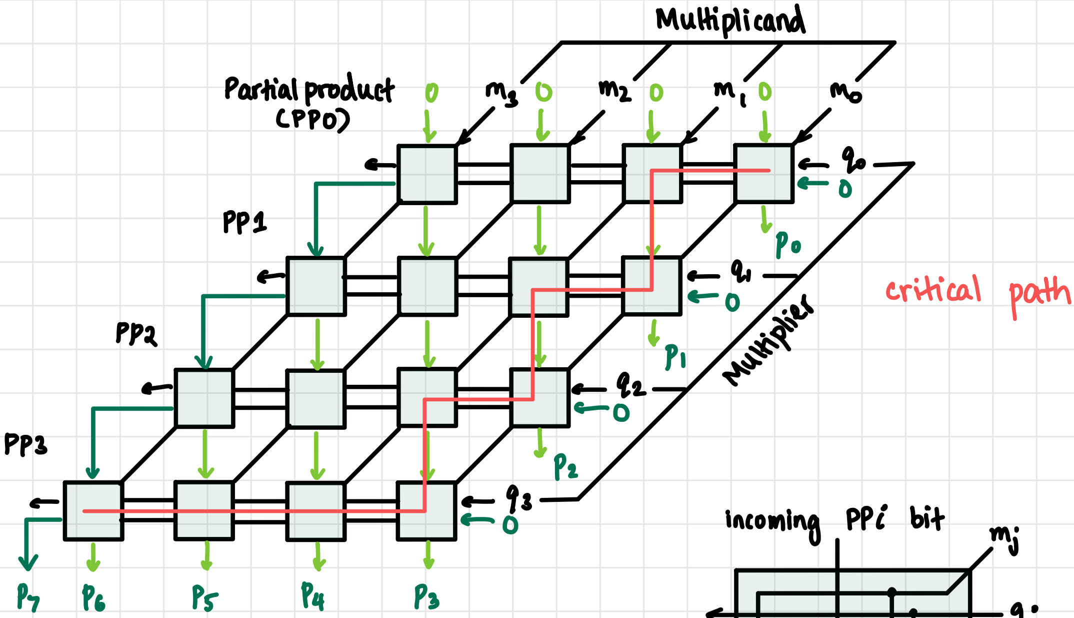
- Using blocks



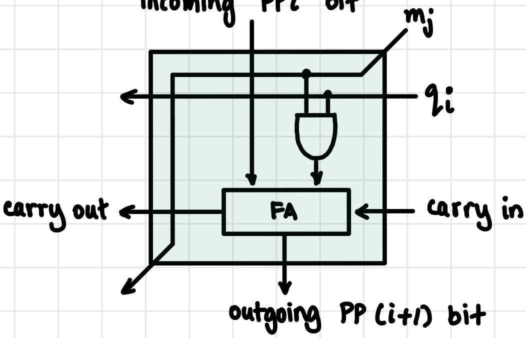
- typical cell (m_j, q_i)



Critical Path Delay



$PP4 = P_7 P_6 \cdots P_0$ = Product



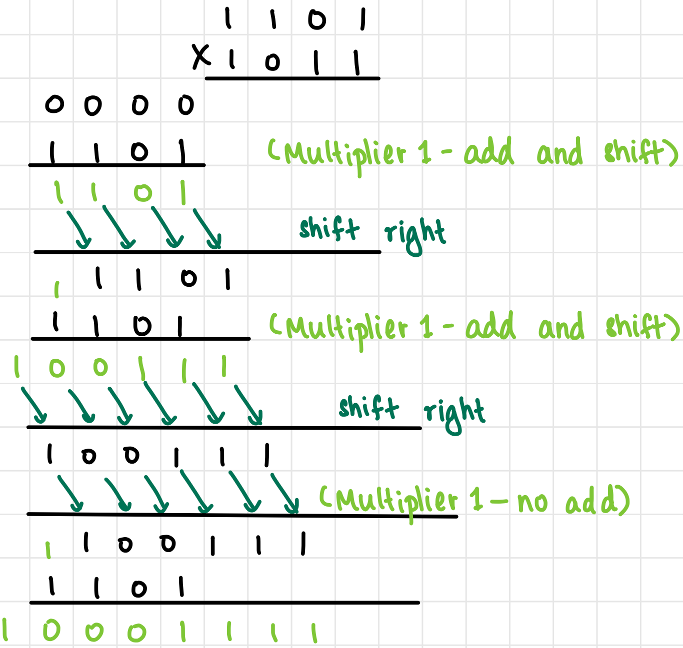
- Assume FA has 2 gate delays and AND gates have 1 gate delay
- Each block: 3 gate delay for carry and outgoing PP bit

$$t_{critical} = 10 \times 3 = 30\, t_g$$

- For m×n multiplication, m×n blocks required
- Hardware intensive ; expensive
- Resources idle

SHIFT-ADD MULTIPLIER

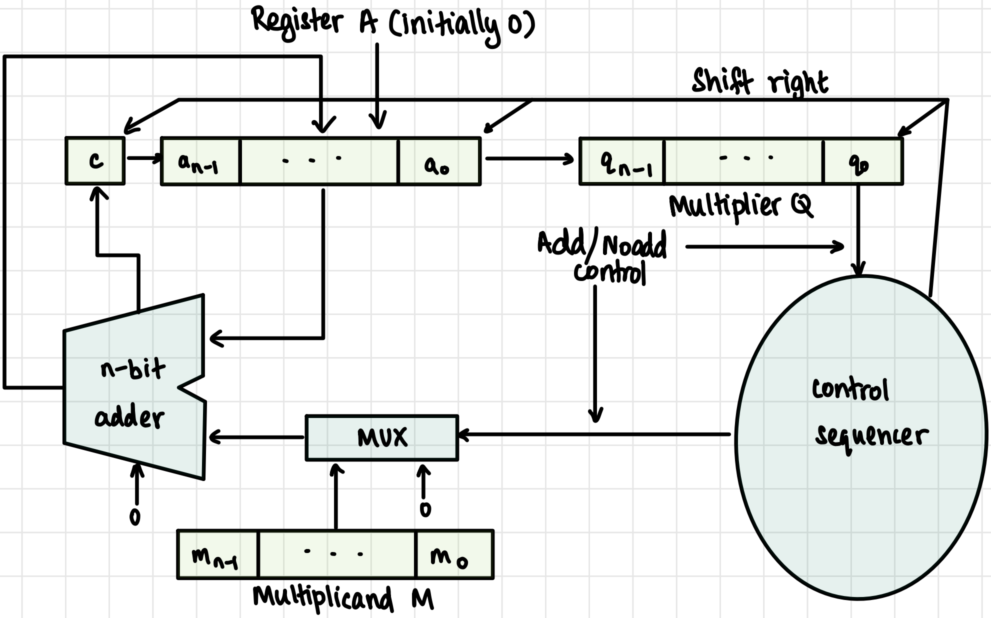
- Multicycle multiplier
- Iterative in nature
- Resources can be reused



- Used for unsigned / positive numbers
- n-bit multiplier requires n cycles

hardware REQUIREMENTS

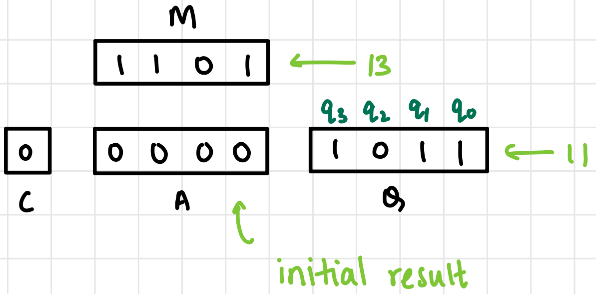
1. Accumulator register — A (stores intermediate / final results)
2. Multiplier register — Q
3. Multiplicand register — M
4. N-bit adder
5. control signals for shift and add



Question 1

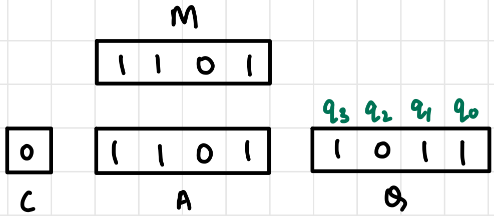
13 x 11

1101 1011

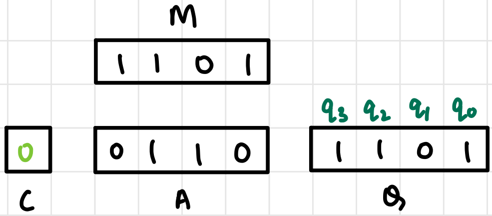


1. $n=1$ iteration

- $q_0 = 1$: add A and M and store in A (Add controller is 1)

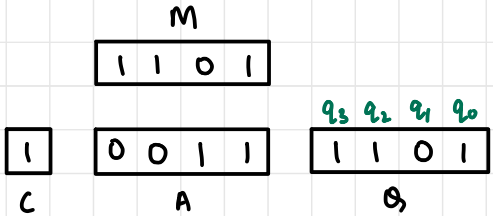


- Shift operation

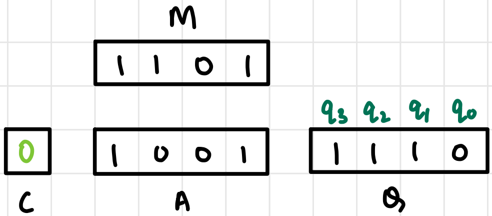


2. $n=2$ iteration

- $q_0 = 1$: add A and M and store in A (Add controller is 1)

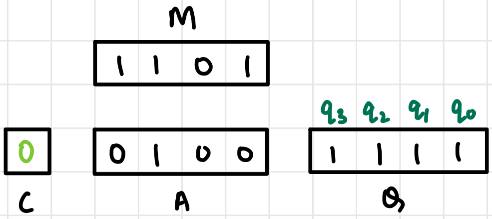


- Shift operation



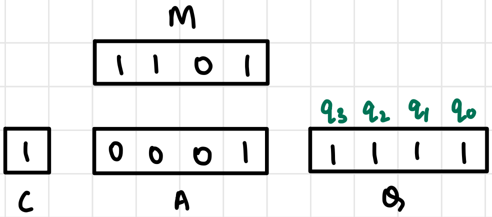
3. n= 3 iteration

- $q_0 = 0$: no add
- only right shift

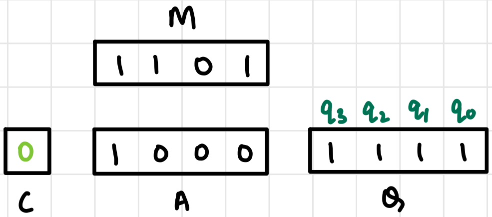


4. n=4 iteration

- $q_0 = 1$: add A and M and store in A (Add controller is 1)



- Shift operation



- Product = 10001111 (143)

SIGNED MULTIPLIER

- The above algorithm works only for unsigned / positive signed numbers

- For it to work for signed numbers, we can perform it in one of two ways

1. Let sign of product $\text{sign}(P) = \text{sign}(M) \overset{\text{XOR}}{\oplus} \text{sign}(Q)$

    - convert M and Q to positive numbers and multiply using shift-add

    - compute sign and convert product accordingly

2. Sign extension on shift

    - for example, M = −3 and Q = +5

M = 1101 (−3)
Q = 0101 (+5)
P = 1111 0001 (−15)

$$
\begin{array}{ccccccccc}
 & & & & & 1 & 1 & 0 & 1 \\
 & & & & \times & 0 & 1 & 0 & 1 \\
\hline
 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
 & 1 & 1 & 1 & 1 & 0 & 1 & & \\
 & 0 & 0 & 0 & 0 & 0 & & & \\
\hline
\textcircled{1} & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
\downarrow & & & & & & & & \\
\text{carry} & & & & & & & &
\end{array}
$$

Question 2

10 x 12 M Q [1010 x 1100]

M

1 0 1 0

	C	A	Q		
	0	0 0 0 0	1 1 0 0	$q_0 = 0$	
only shift					n = 1
	0	0 0 0 0	0 1 1 0	$q_0 = 0$	
only shift					n = 2
	0	0 0 0 0	0 0 1 1	$q_0 = 1$	
add & shift					n = 3
	0	1 0 1 0	0 0 1 1		
	0	0 1 0 1	0 0 0 1	$q_0 = 1$	
add & shift					n = 4
	0	1 1 1 1	0 0 0 1		
	0	0 1 1 1	1 0 0 0		

product : 01111000 = 120

BOOTH ALGORITHM

- can reduce the no. of additions (sequence of 1's)
- Positive & negative (2's complement) no.s treated the same

Algorithm

- In decimal, when multiplying by 9, we use a faster technique
- we do (n x 10) - 1
- Similarly, in binary, suppose we must multiply by 7, we multiply by 8 and subtract 1 ($8 = 2^3$)

Question 3

5 x 7

```
0101 x 0111
0101 x 100-1   ← 100-1 indicates 1000-1 with 1's place as -1
```

NOT TO BE CONFUSED WITH 4-1 AS 100-1; IT IS A SINGLE NUMBER

```
              0  1  0  1
          x   1  0  0 -1    ← Booth recoded multiplier
    '1 1' 1'  1  0  1  1    ← 0101 x -1 → 2's complement
                               with sign extended
     0  0  0  0  0  0
     0  0  0  0  0
     0  1  0  1
carry
out ① 0  1  0  0  0  1  1   ← 35
```

- Booth recoding allows for less number of 1's and increases no. of 0's

- Less number of additions

Transitions

- At i & i-1 position

i i-1
00 → 0 (0-0)
01 → +1 (1-0)
10 → -1 (0-1)
11 → 0 (1-1)

(i-1 - i gives the number)

- Assume Q = 16270

imagine there is a 0 at i = -1

0 0 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0

0 1 0 0 0 0 0 0 -1 0 0 1 0 0 -1 0

- Reduced number of 1's

Worst case

- No sequence of 1's, eg: Q = 85

0 1 0 1 0 1 0 1 0
1 -1 1 -1 1 -1 1 -1 → too many additions

Best Case

- String of 1's

q_0 q^{-1}

0 0 ⟶ no add, shift right with sign extension (0×M)

0 1 ⟶ add M to A (+1×M)

1 0 ⟶ subtract (2's comp) M from A (-1×M)

1 1 ⟶ no add, shift right with sign extension (0×M)

Question 4

M = -3 , Q = -5 multiply using Booth's

M

1 1 0 1

$M^{2's}$ = 0 0 1 1

Q_0

0 0 0 0	1 0 1 1	0
A	Q	Q^{-1}

1) $Q_0 Q^{-1}$ ⟶ 10 (add 2's comp of M to A)

0 0 1 1	1 0 1 1	0
A	Q	Q^{-1}

Shift right with sign extension

Q_0

0 0 0 1	1 1 0 1	1
A	Q	Q^{-1}

2) $Q_0 Q_{-1} \longrightarrow 11$ (shift)

0	0	0	0	1	1	1	$\overset{Q_0}{0}$	1
	A				Q			Q_{-1}

3) $Q_0 Q_{-1} \longrightarrow 01$ (add M to A)

1	1	0	1	1	1	1	0	1
	A				Q			Q_{-1}

Shift right with sign extension

1	1	1	0	1	1	1	1	0
	A				Q			Q_{-1}

4) $Q_0 Q_{-1} \longrightarrow 10$ (add 2's comp of M)

$$\begin{array}{cccccc} & 1 & 1 & & & \\ & 1 & 1 & 1 & 0 \\ + & 0 & 0 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 & 1 \end{array}$$

1	0	0	0	1	1	1	1	1	0
		A				Q			Q_{-1}

Shift right

0	0	0	0	1	1	1	1	1
	A				Q			Q_{-1}

answer = 00001111

= 15

Question 5

-13×11 using Booth

-13 : 10011 ⟶ M

11 : 01011 ⟶ Q

M

1 0 0 1 1

Q $\quad Q_0 \quad Q_{-1}$

0 1 0 1 1 $\quad$ 0

1) Booth encoding of Q

1 -1 1 0 -1 $\quad (-1 + 4 - 8 + 16 = 11)$

2) Column Multiplication

						1	0	0	1	1
					x	1	-1	1	0	-1
		1	1	1	1	10	1			
		0	0	0	0	0	1	1	0	1
		1	1	1	0	0	1	1	0	0
		0	0	1	1	0	1	0	0	0
		1	0	0	1	1	0	0	0	0
negative	1	1	0	1	1	1	0	0	0	1
2's comp		0	1	0	0	0	1	1	1	1

Question 6

13 x −6

13 x − 6

↙ 01101   ↘ 11010 0

0−1+1−10

					0	1	1	0	1
1	1	1	1	1	0	−1	1	−1	0
1	1	1	1	1	0^1	0^1	1	1	0
0	0	0	0	1	1	0	1	0	0
1	1	1	0	0	1	1	0	0	0
1	1	1	0	1	1	0	0	1	0

↓

0 0 0 1 0 0 1 1 1 0 = −78

Question 7

−7 x −4

−7 x −4

↙ 1 0 0 1   ↘ 1 1 0 0 0

0 −1 0 0

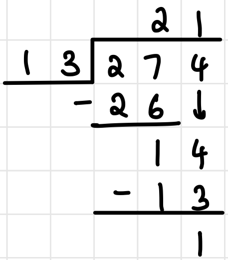
		1	0	0	1
	x	0	−1	0	0
0	1	1	1	0	0

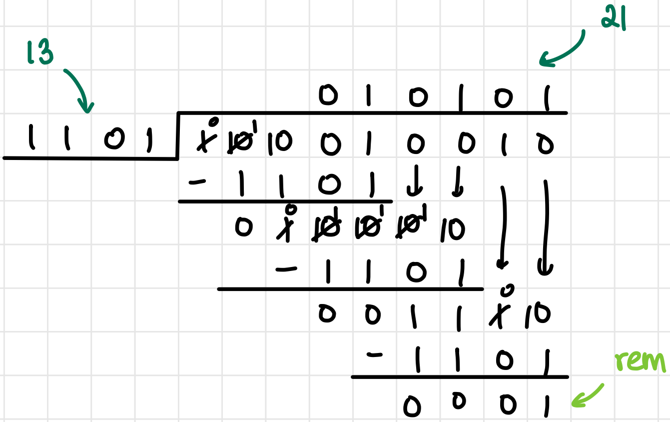
= 28

BINARY DIVISION

- Long division for binary similar to decimal

13 → Divisor (M) (1101)
274 → Dividend (Q) (1 0001 0010)

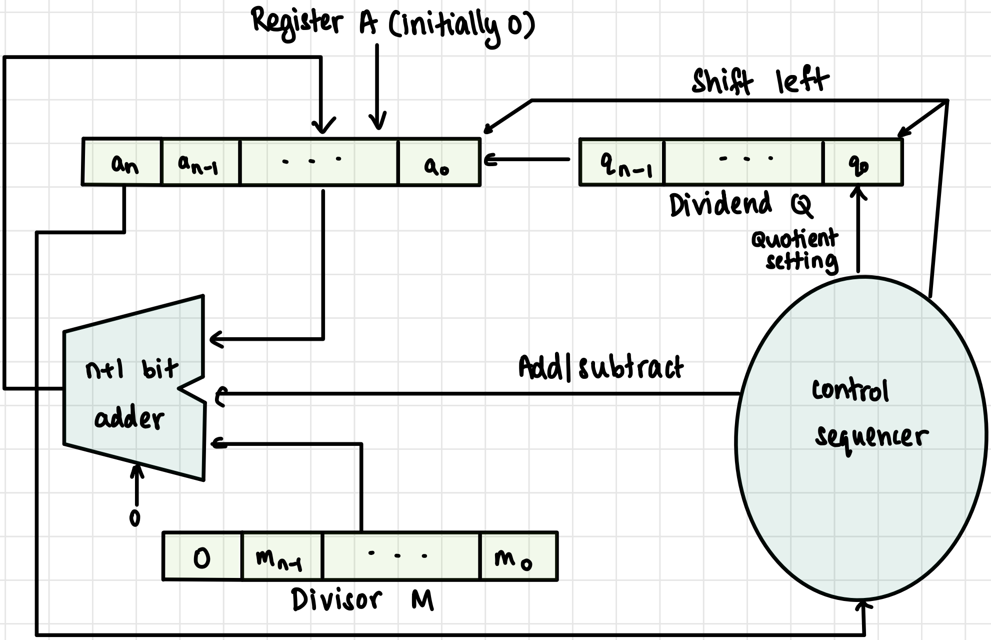




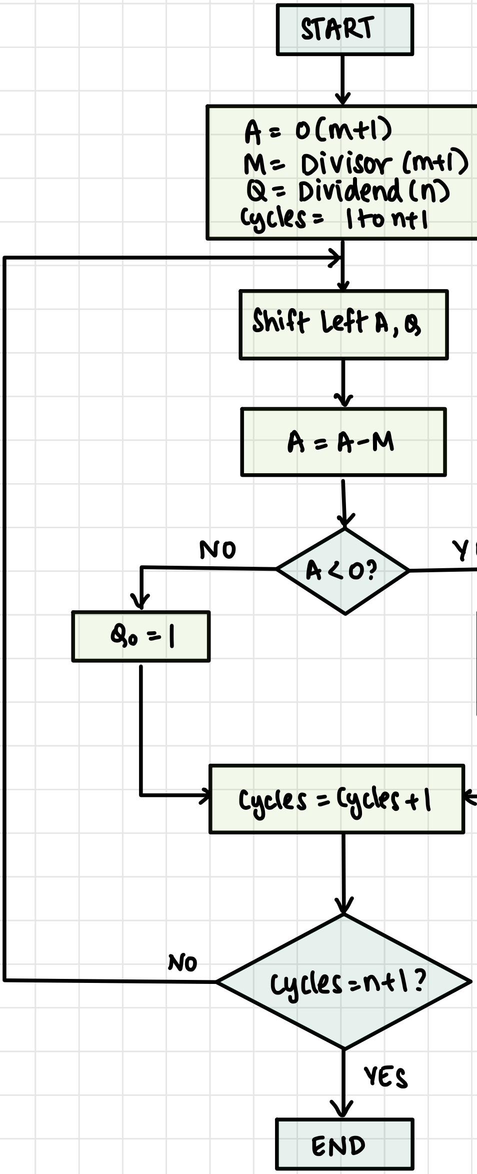
- Subtraction of divisor is performed with dividend
- If remainder is 0 or positive, quotient bit is 1, the remainder is extended by another bit of the dividend to repeat the process
- If remainder is negative, quotient bit is 0 and the dividend is restored by adding back the divisor
- The process is repeated until all the digits in the dividend are considered
- Algorithm —— restoring division algorithm

hardware REQUIREMENTS

1. Accumulator register — A (n+1 bits – remainder)
2. Dividend register — Q
3. Divisor register — M
4. N-bit adder
5. control signals for shift and add/sub



Steps in the Algorithm



Question 8

Show 8 ÷ 3

sign

Divisor = ⓪0011 (M) (3)

Dividend = 1000 (Q) (8)

sign

2's comp of divisor = ①1101 (-3)

M
0 0 0 1 1

S	A	Q
0	0 0 0 0	1 0 0 0

1) Shift left A & Q

S	A	Q
0	0 0 0 1	0 0 0 ?

Subtract M from A

S	A	Q
1	1 1 1 0	0 0 0 ?

Signed bit is 1 → $Q_0 = 0$

Restore prev result by adding M to A

S	A	Q
0	0 0 0 1	0 0 0 0

2) Shift left A & Q

0	0 0 1 0	0 0 0 ?
S	A	Q

Subtract M from A

1	1 1 1 1	0 0 0 ?
S	A	Q

Signed bit is 1 → $Q_0 = 0$
Restore prev result by adding M to A

0	0 0 1 0	0 0 0 0
S	A	Q

3) Shift left A & Q

0	0 1 0 0	0 0 0 ?
S	A	Q

Subtract M from A

0	0 0 0 1	0 0 0 ?
S	A	Q

Signed bit is 0 → $Q_0 = 1$

0	0 0 0 1	0 0 0 1
S	A	Q

4) Shift left A & Q

0	0 0 1 0	0 0 1 ?
S	A	Q

Subtract M from A

1	1 1 1 1	0 0 1 ?
S	A	Q

Signed bit is 1 ⟶ $Q_0 = 0$
Restore prev result by adding M to A

0	0 0 1 0	0 0 1 0
S	A	Q
	remainder = 2	quotient = 2

Question 9

14 ÷ 5 using restoring division

M = 5 = 00101
$M^{2's}$ = -5 = 11011

Q = 14 = 1110
A = 0 = 0000
S = 0

1) Step 1

Left shift:	S=0	A = 0001	Q = 110 ?
Subtract M:	S = ①	A = 1100	Q = 110 ?
Restore & Q_0=0:	S=0	A = 0001	Q = 1100

2) Step 2

Left shift: $S = 0$ $A = 0011$ $Q = 100?$

Subtract M: $S = ①$ $A = 1110$ $Q = 100?$

Restore & $Q_0 = 0$: $S = 0$ $A = 0011$ $Q = 1000$

3) Step 3

Left shift: $S = 0$ $A = 0111$ $Q = 000?$

Subtract M: $S = ⓪$ $A = 0010$ $Q = 000?$

$Q_0 = 1$: $S = 0$ $A = 0010$ $Q = 0001$

4) Step 4

Left shift: $S = 0$ $A = 0100$ $Q = 001?$

Subtract M: $S = ①$ $A = 1111$ $Q = 001?$

Restore & $Q_0 = 0$: $S = 0$ $A = \underbrace{0100}_{\text{remainder} = 4}$ $Q = \underbrace{0010}_{\text{quotient} = 2}$

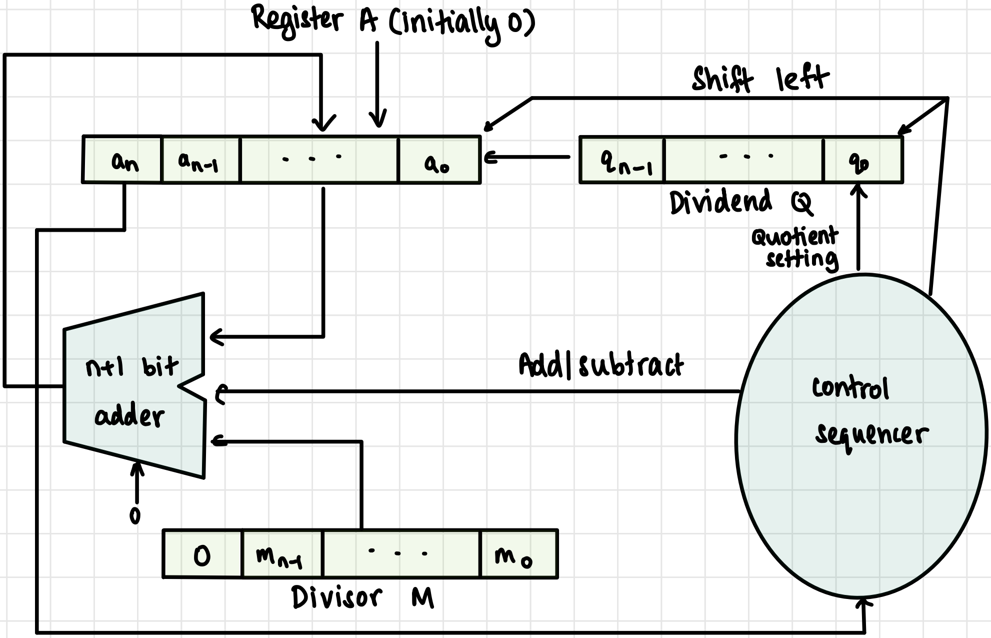
OBSERVATIONS

- Used for unsigned/positive signed binary numbers
- Restorations require significant number of subtractions which can be avoided
- If A is positive, we do $\boxed{2A-M}$ (shift left → 2× number)
- If A is negative, we restore A (A+M) and then left shift and subtract M → $2(A+M)-M = \boxed{2A+M}$

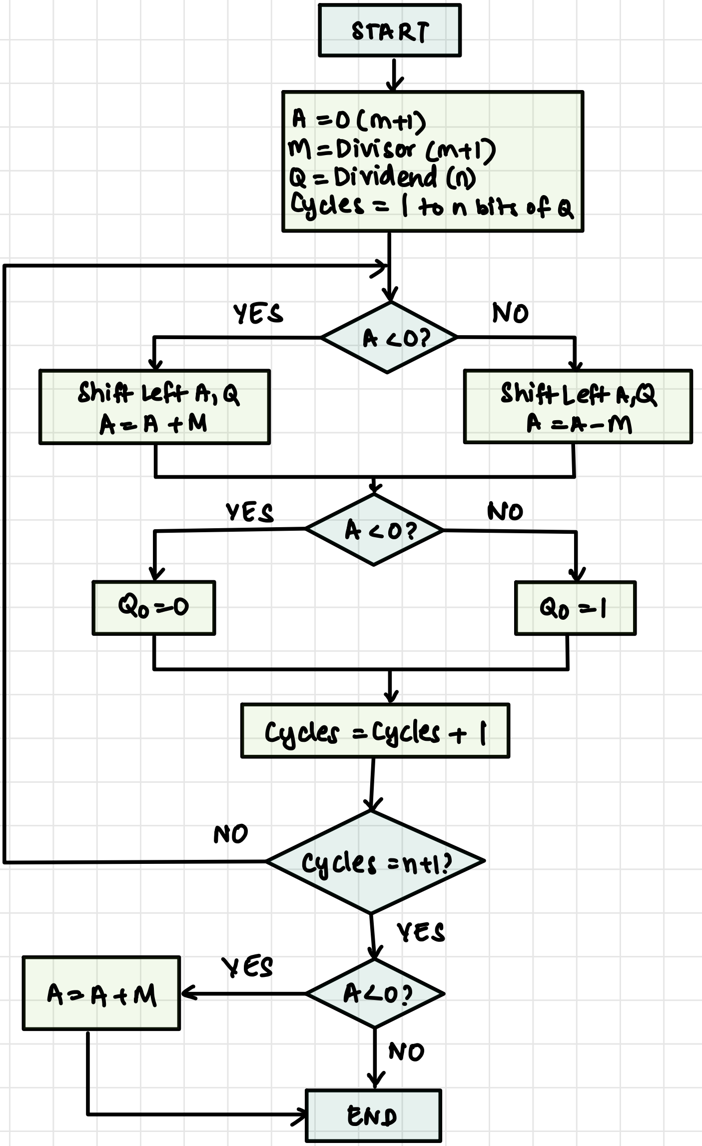
NON-RESTORING DIVISION

hardware REQUIREMENTS

1. Accumulator register — A (n+1 bits – remainder)
2. Dividend register — Q
3. Divisor register — M
4. N-bit adder
5. control signals for shift and add/sub



Algorithm


START
A = 0 (m+1)
M = Divisor (m+1)
Q = Dividend (n)
Cycles = 1 to n bits of Q
A < 0?
YES
NO
Shift Left A, Q
A = A + M
Shift Left A,Q
A = A − M
A < 0?
YES
NO
Q0 = 0
Q0 = 1
Cycles = Cycles + 1
Cycles = n+1?
NO
YES
A < 0?
YES
A = A + M
NO
END

Question 10

Show 8 ÷ 3

$Q = 1000$

$M = 0\ 0011$

$M^{2's} = 1\ 1101$

sign

M
0 0 0 1 1

0	0 0 0 0	1 0 0 0
S	A	Q

1) S is 0 → shift left & subtract M

0	0 0 0 1	0 0 0 ?
S	A	Q

1	1 1 1 0	0 0 0 ?
S	A	Q

S is 1 → $Q_0 = 0$

1	1 1 1 0	0 0 0 0
S	A	Q

2) S is 1 → shift left & add M

1	1 1 0 0	0 0 0 ?
S	A	Q

1	1 1 1 1	0 0 0 ?
S	A	Q

S is 1 → $Q_0 = 0$

S	A	Q
1	1 1 1 1	0 0 0 0

3) S is 1 → shift left & add M

S	A	Q
1	1 1 1 0	0 0 0 ?

S	A	Q
0	0 0 0 1	0 0 0 ?

S is 0 → $Q_0 = 1$

S	A	Q
0	0 0 0 1	0 0 0 1

4) S is 0 → shift left & subtract M

S	A	Q
0	0 0 1 0	0 0 1 ?

S	A	Q
1	1 1 1 1	0 0 1 ?

S is 1 → $Q_0 = 0$

S	A	Q
1	1 1 1 1	0 0 1 0

5) S is 1 ⟶ add M to A

$$\underset{S}{\boxed{0}} \qquad \underbrace{\underset{A}{\boxed{0\ 0\ 1\ 0}}}_{\text{remainder}=2} \qquad \underbrace{\underset{Q}{\boxed{0\ 0\ 1\ 0}}}_{\text{quotient}=2}$$

OBSERVATIONS

- Used for unsigned/positive signed binary numbers
- For signed numbers, convert to positive and adjust sign later
- Resultant sign will be XOR of divisor and dividend signs

Worst case in terms of no. of add/subs ?

- n+1 steps (if final step is negative)

Question 11

15 ÷ 5 (non-restoring)

A = 0000 M = 00101 $M^{2's}$ = 11011 Q = 1111

Step 1

Left shift : S = 0 A = 0001 Q = 111 ?
Subtract M : S = 1 A = 1100 Q = 111 ?
S = 1, Q_0 = 0: S = 1 A = 1100 Q = 1110

Step 2

Left shift:	S = 1	A = 1001	Q = 110?
Add M:	S = 1	A = 1110	Q = 110?
S = 1, $Q_0 = 0$:	S = 1	A = 1110	Q = 1100

Step 3

Left shift:	S = 1	A = 1101	Q = 100?
Add M:	S = 0	A = 0010	Q = 100?
S = 0, $Q_0 = 1$:	S = 0	A = 0010	Q = 1001

Step 4

Left shift:	S = 0	A = 0101	Q = 001?
Subtract M:	S = 0	A = 0000	Q = 001?
S = 0, $Q_0 = 1$:	S = 0	$\underbrace{A = 0000}_{\text{remainder = 0}}$	$\underbrace{Q = 0011}_{\text{quotient}}$

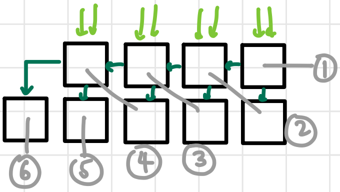
WALLACE TREE MULTIPLIER

Carry-Save Addition

- Addition of 3 binary numbers normally

$$\begin{array}{cccccc}
 & 10 & 10 & 1 & \\
 & 0 & 1 & 1 & 1 \\
+ & 1 & 0 & 1 & 1 \\
+ & 0 & 1 & 1 & 0 \\
\hline
1 & 1 & 0 & 0 & 0
\end{array}$$

- one n-bit adder
- one n+1-bit adder
- result upto n+2 bits



Time Delay

- n-bit RCA takes $n\,t_{FA}$
- total time = $(n+2)\,t_{FA}$

parallelly

- if parallel prefix adder used, factor of 2 still present
- lower bound: time taken for 2 numbers

Carry-Save

(without incoming carry)

$$\begin{array}{rccccc}
 & & 0 & 1 & 1 & 1 \\
 & + & 1 & 0 & 1 & 1 \\
 & + & 0 & 1 & 1 & 0 \\
\hline
\text{Sum} & & 1' & 0' & 1 & 0 \\
\text{Carry} & 1 & 0 & 1 & 1 & 1 \\
\text{carry left shift} & 0 & 1 & 1 & 1 & 0 \\
\hline
 & \multicolumn{5}{c}{\underbrace{1\quad 1\quad 0\quad 0\quad 0}_{n+1\text{-bit number}}}
\end{array}$$

add

] full adder results

] left shift carry bits

© vibha's notes 2020

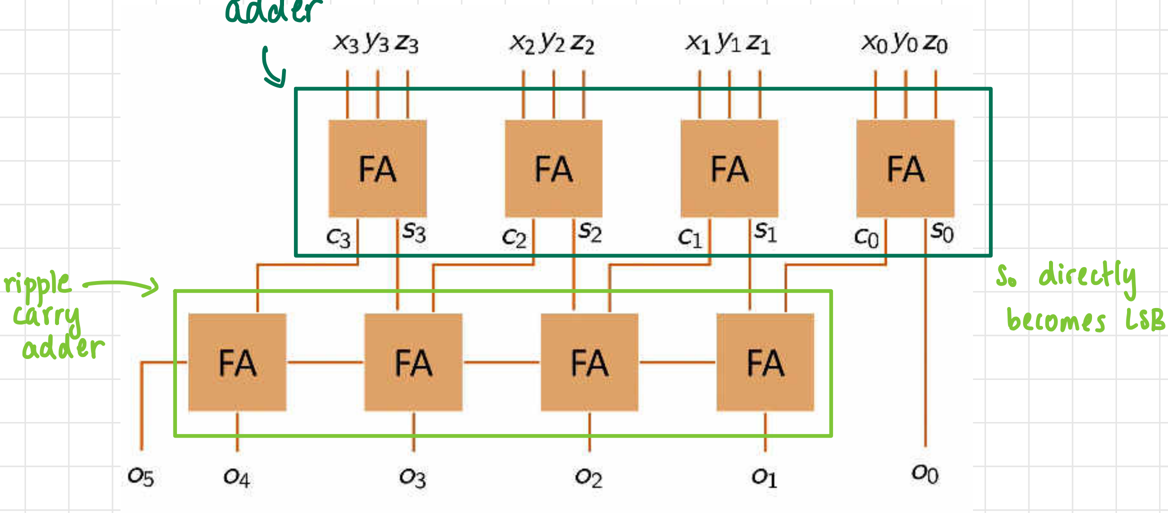
- Instead of left-shifting carry, we directly write the LSB of sum as the LSB of result

- Add n-1 bits of sum to n bits of carry

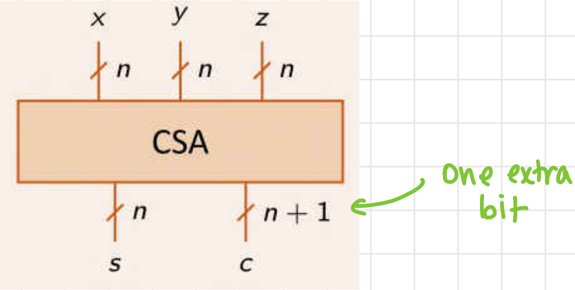
Time Delay

- $n t_{FA}$ for the second step
- For the first step, n full adders can add in parallel without having to propagate
- $(n+1) t_{FA}$ → only t_{FA} more than the time taken to add two numbers

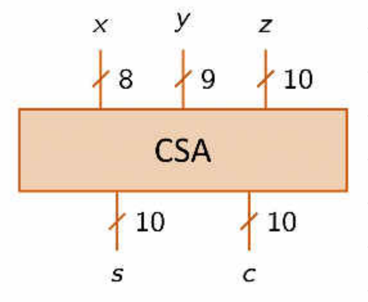
carry save adder



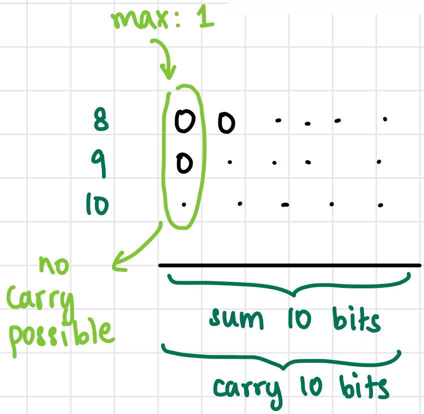
Symbol



Carry-Save Adder with Different Sized Inputs



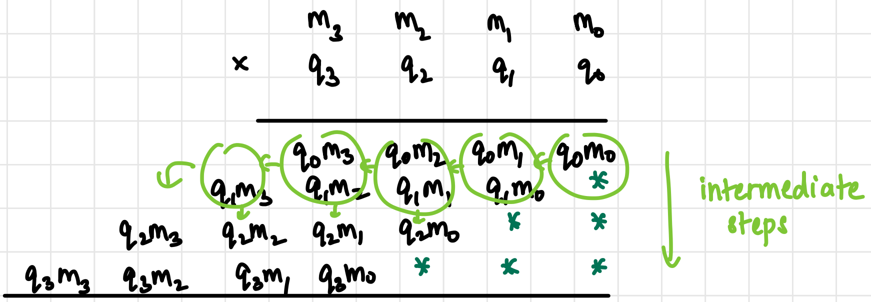
∴ needs 10 FAs



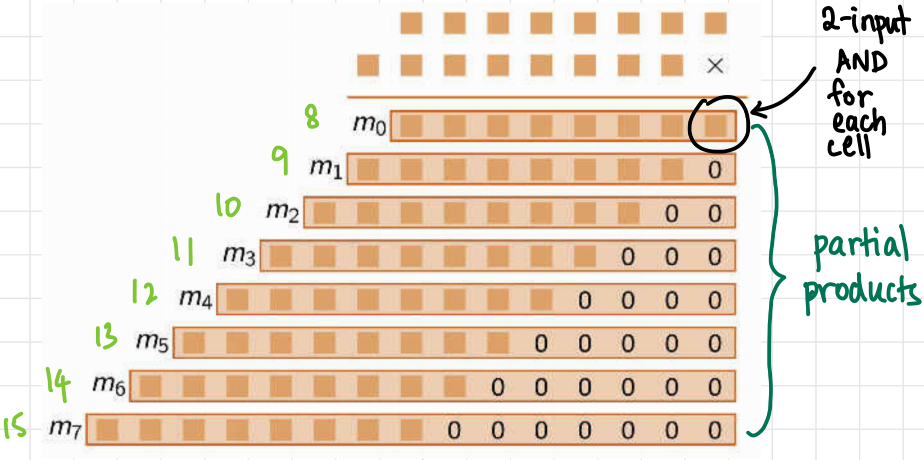
∴ for CSA of n, n-1, n-2 bits, sum and carry are n-bits

Wallace Tree Multiplier

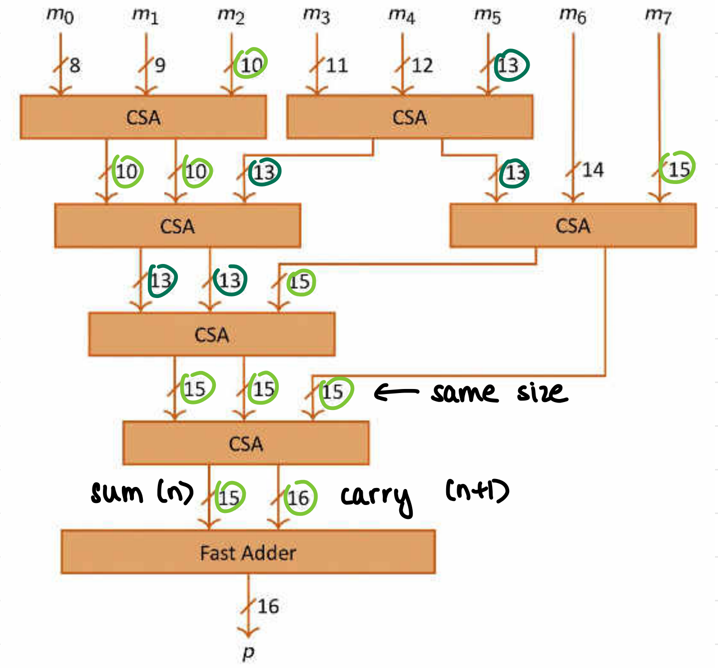
Ripple Carry Multiplier

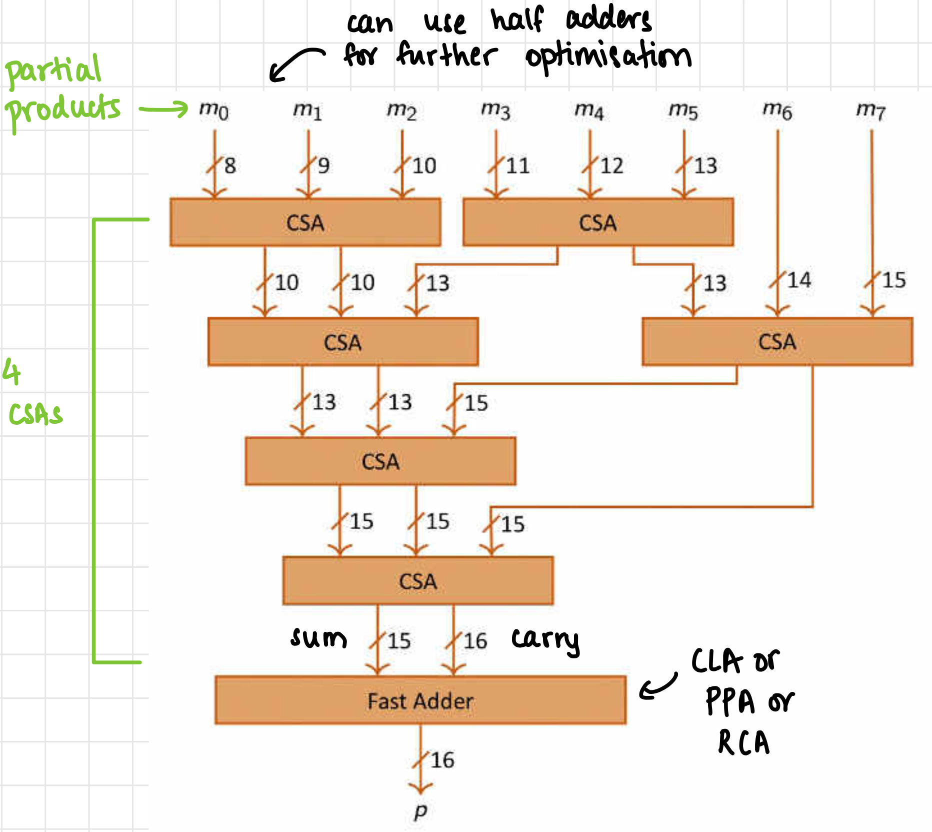


8-bit Multiplier



- m_i has size $(8+i)$ bits — using AND gate
- Add all the partial products





can use half adders for further optimisation

partial products

4 CSAs

CLA or PPA or RCA

- Area is not very low

Time Delay

- t_{Adder} = delay of fast adder
- t_{AND} = time to compute partial products
- t_{FA} = time for full adder
- 4 t_{FA} time for CSAs
- for 8-bit multiplier

$$\boxed{t_{cp} = t_{AND} + 4t_{FA} + t_{Adder}}$$

Compare the area and time performance of the Wallace Tree Multiplier with that of the Shift-Add Multiplier

- Try ripple carry and parallel prefix adders in both (8x8)

	Wallace Tree Multiplier	Shift-add Multiplier
time	for 8-bit x 8-bit number $t = t_{AND} + 4t_{FA} + t_{adder}$	for each iteration 1. add 2 8-bit no.s $\longrightarrow t_{adder}$ 2. left shift by one $\longrightarrow t_{2:1\,MUX}$ $t = 8\,t_{adder} + 8\,t_{2:1\,MUX}$ ↖ shift reg?
space	each CSA = $4\,a_{FA}$ ∴ for 6 CSAs = $24\,a_{FA}$ $a = 24\,a_{FA} + a_{adder}$	16x2:1 MUXes for shifter excluding registers $a = 16\,a_{2:1\,MUX} + a_{adder}$

FLOATING POINT NUMBERS

floating point representation

Question 12

Convert 0110.1100 to decimal

0	1	1	0	.	1	1	0	0
2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}
8	4	2	1		1/2	1/4	1/8	1/16

= 6.75

Question 13

Convert 6.75 to binary

6 ⟶ 0110

0.75 × 2 = 1.50 1

0.50 × 2 = 1.00 1

0.00 × 2 = 0.00 0

= 0.110

∴ 6.75 = 0110.1100

Question 14

Convert 0.625 to binary

0.625 x 2 = 1.250 → 1
0.250 x 2 = 0.500 → 0
0.500 x 2 = 1.000 → 1
0.000 x 2 = 0.000 → 0

= 0.1010

Representation of Real Numbers

1. Fixed point notation
   - decimal point fixed
   - limited by space

2. Floating point notation
   - allows for more numbers

Question 15

Represent -7.5 using fixed point notation (4 integer bits and 4 fraction bits)

+7.5 = 0111.1000 ← decimal point is implied as size is known beforehand

2's complement = 1000.1000

Question 16

Perform 7.5 − 0.625 using fixed point notation

7.5 = 0111. 1000
+0.625 = 0000. 1010
−0.625 = 1111. 0110

7.5 + (−0.625)

$$\begin{array}{cccccccccc}
 & 1 & 1 & 1 & & & & & & \\
 & 0 & 1 & 1 & 1 & . & 1 & 0 & 0 & 0 \\
\text{ignore} \; + & 1 & 1 & 1 & 1 & . & 0 & 1 & 1 & 0 \\ \hline
\searrow \text{①} & 0 & 1 & 1 & 0 & . & 1 & 1 & 1 & 0
\end{array}$$

6. (0.5 + 0.25 + 0.125)
6. (0.875)

= 6.875

Question 17

Perform 8.9375 + 8.3125

$$\begin{array}{rcr}
 & & 1\;1\;1\;1 \\
8.9375 & = & 1000.1111 \\
+\; 8.3125 & = & 1000.0101 \\ \hline
 & & 10001.0100
\end{array} = 1.25$$

overflow

due to limited space

Floating Point Representation

$$\pm \underset{\substack{\uparrow\\ \text{mantissa}}}{M} \times \underset{\substack{\nwarrow\\ \text{base}}}{B}^{\pm E \;\leftarrow\; \text{exponent}}$$

- should be normalised → use one non-zero digit as integer part
    - ↳ Decimal : 1-9
    - ↳ Binary : 1
- Example: 2.34×10^{-12} is normalised, 0.342×10^{9} is not normalised (scientific notation)
- IEEE 754-2008 standard for floating point numbers for universality
- Defines four representations

    1. Single precision — 32 bits
    2. Double precision — 64 bits
    3. Extended double precision — 10 bytes / 80 bits
    4. Quadruple precision — 16 bytes / 128 bits

    - Three parts

        1. Sign bit
        2. Exponent bits
        3. Mantissa or significand bits

Single Precision

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exponent								fraction/mantissa																						
1 bit	8 bits								23 bits																						

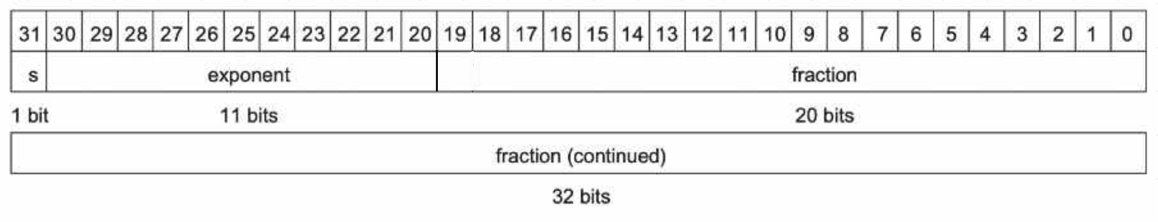
called biased exponent

0 : positive
1 : negative

$$1.\text{mantissa} \times 2^{\text{exponent}}$$

Double Precision

$$BE = E + 2^{11-1} - 1 = E + 1023$$



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exponent											fraction																			
1 bit	11 bits											20 bits																			
fraction (continued)																															
32 bits																															

- mantissa : 52 bits
- exponent : 11 bits ← biased exponent

Biased Exponent

- BE = Bias + Exponent
- Actual exponent = Biased exponent - Bias
- BE for SP number = 8 bits (0-255)
- BE = 0 and BE = 255 are reserved for special use ⇒ 1 to 254 are used for normalised FPNs

- Bias $= 2^{n-1} - 1 = 2^{8-1} - 1 = 2^{7} - 1 = 127$

- Range of actual values:

  min : $1 - 127 = -126$
  max : $254 - 127 = 127$
  range : -126 to 127

- Bias removes the need for 2's complement exponents

- FP representation

$$\boxed{N = (-1)^{s} * (1 + 0.M) * 2^{(BE - Bias)}}$$

Question 18

What FPN does this represent?

	4 3	6 4 0 0 0 0
0	100 0011 0	110 0100 0000 0000 0000 0000
s	e	m
1	8	23 → 32

hex: 0x 43640000

sign: 0
BE : $10000110 = 6 + 128 = 134$
$E = 134 - 127 = 7$
$m = 1.110\ 0100\ 0000\ 0000\ 0000\ 0000$
$= 1 + 2^{-1} + 2^{-2} + 2^{-5} = 1.78125$

1 is implicit

$$N = 1.78125 \times 2^{7} = 228$$

Question 19

What is the decimal value of the given number?

0x BE20 0000

binary

1	011 1110 0	010 0000 0000 0000 0000 0000
s	e	m

s = 1 → negative

BE = 124 ⇒ E = 124 − 127 = −3

$m = 1.010\ 0000 \ldots$

$= 1.25$

$N = -1.25 \times 2^{-3} = -0.15625$

Question 20

Represent the following decimal as binary SPE IEEE-754

-58.25_{10}

s = 1

58 = 0011 1010

0.25 = 0.01

$58.25_{10} = 111\,010.01_2$

Normalise

$$\overset{\text{implicit}}{1}.\underbrace{1101001}_{} \times 2^5$$

m= 1101001

actual E= 5

BE = 127+5 = 132 = 1000 0100

Single precision

1	100	0010	0	110	1001	0000	0000	0000	0000
	C	2		6	9	0	0	0	0

0x C2690000

Question 21

Represent the following decimal as binary DPE IEEE-754

-58.25_{10}

s = 1

$58.25_{10} = 111010.01_2 = 1.1101001 \times 2^5$

m= 1101001

actual E = 5

biased E = $5 + 2^{n-1} - 1$ = 5+1023 = 1028

s = 1
e = 100 0000 0100
m = 1101 0010 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000

0x C04D 2000 0000 0000

Question 22

Convert 7.875 to SPE IEEE-754

7 = 111
0.875 = 0.5 + 0.25 + 0.125 = 0.111

7.875 = 111.111

$= 1.11111 \times 2^2$

m = 11111
AE = 2 ⇒ BE = 129 = 1000 0001
s = 0

0	100 0000 1	111 1100 0000 0000 0000 0000

0x 20FC0000

Question 23

Convert 0.1875 to IEEE-754 SPE format

$0.1875 = 0 + 1/8 + 1/16 = 0.0011$
$= 1.1 \times 2^{-3}$

$s = 0$
$m = 1$
$e = -3 + 127 = 124 = 0111\ 1100$

0	011 1110 0	100 0000 0000 0000 0000 0000

0x 3E40 0000

Special Cases – SPNs

Min value

$1 + \text{fraction} = 1.0000\ 0000 \ldots$
$\text{exponent} = 1 - 127 = -126$ (min = 1 as 00000000 is reserved)

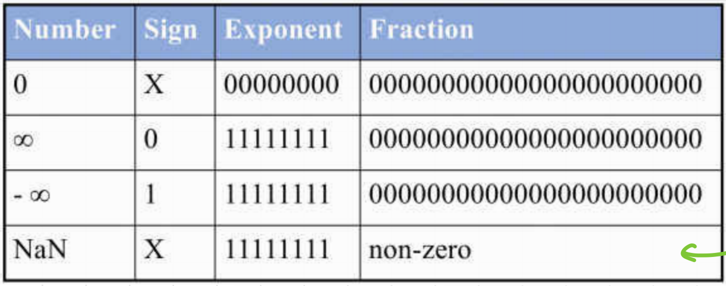
$$= \pm 1.00 \times 2^{-126} = \pm 1.17549 \times 10^{-38}$$

Max value

$1 + \text{fraction} = 1.111111\ldots \approx 2$
$\text{exponent} = 254 - 127 = 127$ (max ≠ 11111111)

$$= 2 \times 2^{127} = 2^{128} = \pm 3.4028 \times 10^{38}$$

Special Cases



Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

division by zero

Overflow & Underflow

Overflow: number is too big to represent

Underflow: number is too small to represent

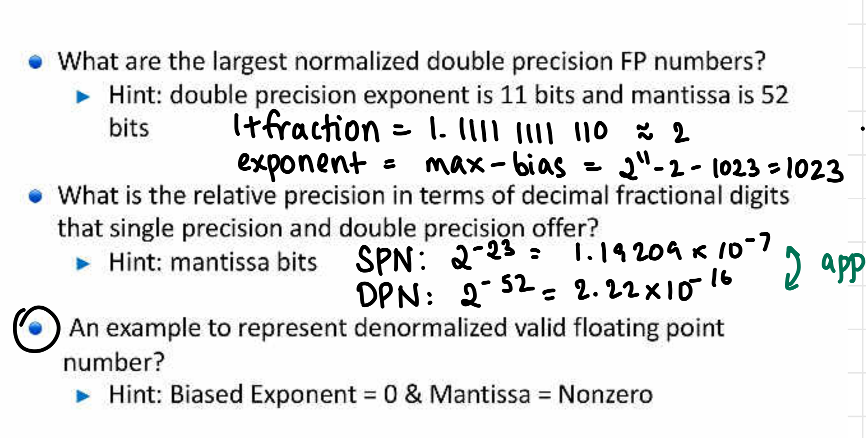
Rounding Modes

1) Down
2) Up
3) Towards zero
4) Nearest

Question 24

Round 1.100101 (1.578125) to only 3 fraction bits

1. Down: 1.100
2. Up: 1.101
3. Towards zero: 1.100
4. To nearest: 1.101 (1.625 closer than 1.5)



- What are the largest normalized double precision FP numbers?
  - Hint: double precision exponent is 11 bits and mantissa is 52 bits

$1+\text{fraction} = 1.1111\ 1111\ 110 \approx 2$

$\text{exponent} = \text{max} - \text{bias} = 2^{11} - 2 - 1023 = 1023$

$\therefore \text{max} = 2 \times 2^{1023}$

$= 2^{1024}$

- What is the relative precision in terms of decimal fractional digits that single precision and double precision offer?
  - Hint: mantissa bits

SPN: $2^{-23} = 1.19209 \times 10^{-7}$

DPN: $2^{-52} = 2.22 \times 10^{-16}$

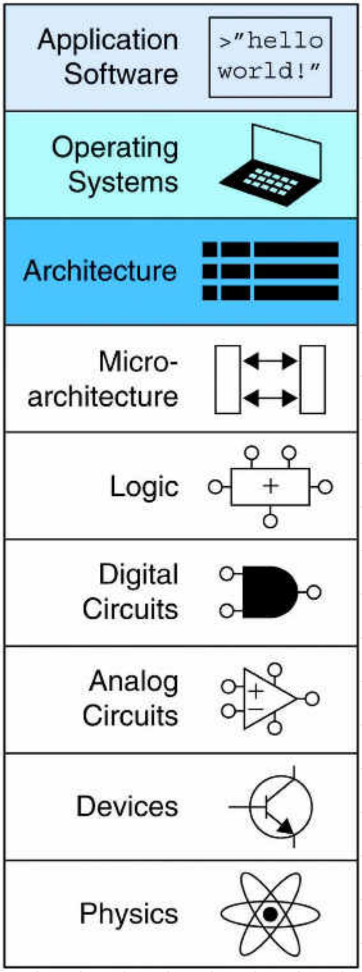
} approx double?

- An example to represent denormalized valid floating point number?
  - Hint: Biased Exponent = 0 & Mantissa = Nonzero

TODO

COMPUTER ARCHITECTURE

Levels of Abstraction



Unit 4 (chapter 6)

→ assembly language, etc

Unit 5 (chapter 7)

→ reg files, ALUs (lab)

→ logic structures

→ gates

→ ECE chem cycle

→ transistors

→ PHY physics cycle

Assembly Language

- Instructions are commands in a computer's language
- Computer-readable format of 1's and 0's is machine language
- Assembly language is a human-readable format of instructions
- Assembly language different for different architectures (x86 — RISC — reduced instruction set computing, ARM)
- MIPS architecture : developed by John Hennessy & his colleagues at Stanford, used in many commercial systems (Nintendo, Cisco, Silicon Graphics)
- Hennessy & Patterson — Turing award

Underlying Design Principles

1. Simplicity favours regularity
2. Make the common case fast
3. Smaller is faster
4. Good design demands good compromises

instructions

Addition

- C Code

  a = b + c;

- MIPS assembly code

  add a, b, c

- add: mnemonic indicates operation to perform
- b, c: source operands (on which the operation is to be performed)
- a : destination operand

Subtraction

- C Code

  a = b - c;

- MIPS assembly code

  sub a, b, c

- sub: mnemonic
- b, c: source operands
- a: destination operand

Complex Combination of Add & Sub

- C code

```
a = b + c - d;
```

- MIPS Assembly code

temporary

```
add  t, b, c   # t = b + c
sub  a, t, d   # a = t - d
```

- MIPS is a RISC (reduced instruction set computer) with a small number of simple instructions
- Intel x86 is a CISC (complex instruction set computer) with more complex instructions

#1 Simplicity favours regularity

- consistent instruction format
- same number of operands (2 source, one dest)
- easier to encode and handle in hardware

#2 Make the common case fast

- MIPS - only commonly used instructions
- complex instructions performed using multiple simple instructions

Operand Location

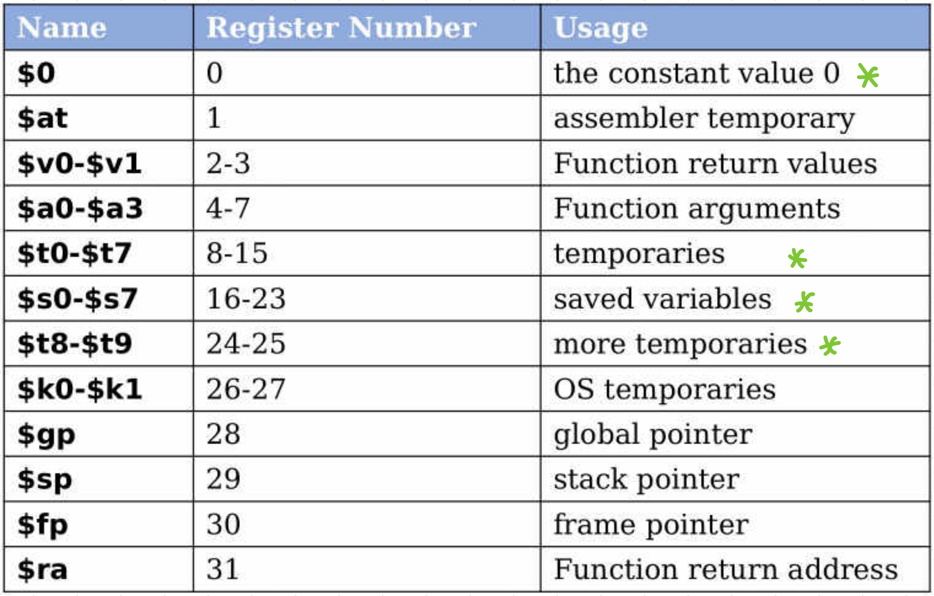
- register (part of microprocessor) - faster
- memory (not part of microprocessor) - slower
- constants (also called immediates) ← used in DDCO lab (both)

#3 Smaller is faster

- MIPS uses only small number of registers

1. REGISTER

- In lab, we implemented 8 16-bit registers in reg-alu
- MIPS has 32 32-bit registers
- Registers are faster than memory
- MIPS called 32-bit architecture because it operates on 32-bit data



Name	Register Number	Usage
$0	0	the constant value 0 *
$at	1	assembler temporary
$v0-$v1	2-3	Function return values
$a0-$a3	4-7	Function arguments
$t0-$t7	8-15	temporaries *
$s0-$s7	16-23	saved variables *
$t8-$t9	24-25	more temporaries *
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	Function return address

Registers

- for registers, we use $ before the name of the register
- example : $0 ⟶ register zero or dollar zero

Registers Used for Specific purposes

- $0 always stores constant value 0 (read-only), used for initialising variables with 0 without having to load from memory
- saved registers $s0 - $s7 are used to hold variables (not temporary)
- temporary registers $t0-$t9 to store intermediate values during a larger computation

Instructions with Registers

- c code - add instruction

```
a = b + c;
```

- MIPS assembly code

```
$s0 = a, $s1 = b, $s2 = c

add $s0, $s1, $s2
```

Question 25

Using only the instructions learnt so far (add and sub), write a program to left shift a number by 3 bit positions. How many registers do you need?

- To shift left once, you multiply by 2
- Same as adding number to itself.
- Repeat thrice

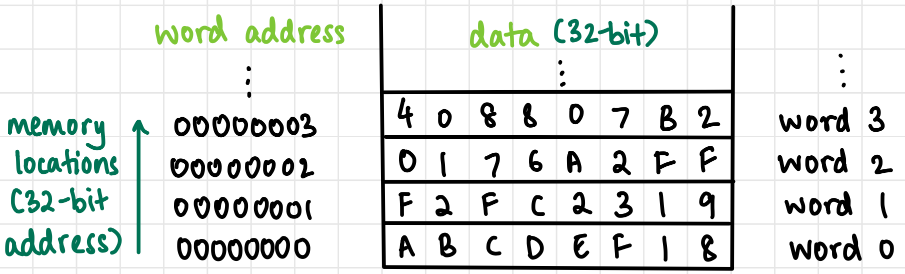
```
# $s0 = a, $s1 = b

add $t0, $s1, $s1
add $t1, $t0, $t0
add $s0, $t1, $t1
```

2 save registers

2 temp registers

2. MEMORY

- Too much data to fit into 32 registers (slow but large)
- Commonly used variables kept in registers
- Main memory
- Sequence of 32-bit words



word address	data (32-bit)	
⋮	⋮	⋮
00000003	4 0 8 8 0 7 B 2	word 3
00000002	0 1 7 6 A 2 F F	word 2
00000001	F 2 F C 2 3 1 9	word 1
00000000	A B C D E F 1 8	word 0

© vibha's notes 2020

Instructions to Access Memory

READ MEMORY

- Memory read called load
- Fetching from memory and storing in register in microprocessor (loading from memory)
- Mnemonic: load word (lw) — op code
- Format: $\texttt{lw } \underbrace{\$s0}_{\text{destination}}, \underbrace{5(\$t1)}_{\text{after comma: address to be fetched from}}$

  $t1 contains base address of memory
- Address calculation:
  - * add base address ($t1) to the offset (5)
  - * address = ($t1+5) contents of $t1 added to 5
  - * fetch data stored in address
- Result
  - * $s0 holds the value at address ($t1+5)
- Any register can be used as base address

Question 26

Read a word of data at memory address 1 into $s3

memory locations (32-bit address) ↑

word address	data (32-bit)	
⋮	⋮	⋮
00000003	4 0 8 8 0 7 B 2	word 3
00000002	0 1 7 6 A 2 F F	word 2
00000001	F 2 F C 2 3 1 9	word 1
00000000	A B C D E F 1 8	word 0

always contains zero

- address = ($0 + 1)
- $s3 = 0xF2FC2319 after load

Assembly code

```
lw $s3, 1($0) # read memory word 1 into $s3
```

WRITE MEMORY

- Memory write called store
- Mnemonic : store word (sw)
- Format : `sw $t1, 0x5($t0)`

offset can be written in decimal (default) or hex

- Store the value in $t1 into memory address ($t0 + 0x5)

Question 27

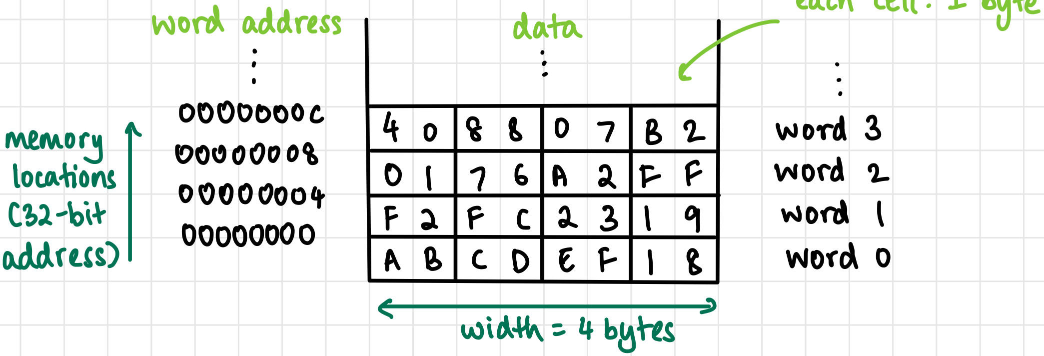
Write (store) the value in $t4 into memory address 7

- add 0x7 to base address: ($0 + 0x7)

Assembly code

```
sw $t4, 0x7($0)    # write the value in $t4
                   # to memory location 7
```

Byte Addressable Memory



⋮		⋮			⋮
0000000C	4 0	8 8	0 7	B 2	word 3
00000008	0 1	7 6	A 2	F F	word 2
00000004	F 2	F C	2 3	1 9	word 1
00000000	A B	C D	E F	1 8	word 0

- Each data byte has a unique address
- Load/ store single bytes (lb, sb)
- 32-bit word = 4 bytes, so word address increments by 4 bits (think long int in C)

Question 28

Load a word of data at memory address 4 into $s3

memory locations (32-bit address) ↑

word address	data (32-bit)				
⋮	⋮				⋮
0000000C	40	88	07	B2	word 3
00000008	01	76	A2	FF	word 2
00000004	F2	FC	23	19	word 1
00000000	AB	CD	EF	18	word 0

$s3 = 0xF2FC2319 after load

Assemby Code

comments start with #

```
lw $s3, 4($0) # read word at address 4 into $s3
```

Question 29

Write (store) the value in $t7 into memory address 0x2C

```
sw $t7, 0x2C($0)  # store $t7 into 0x2C
```

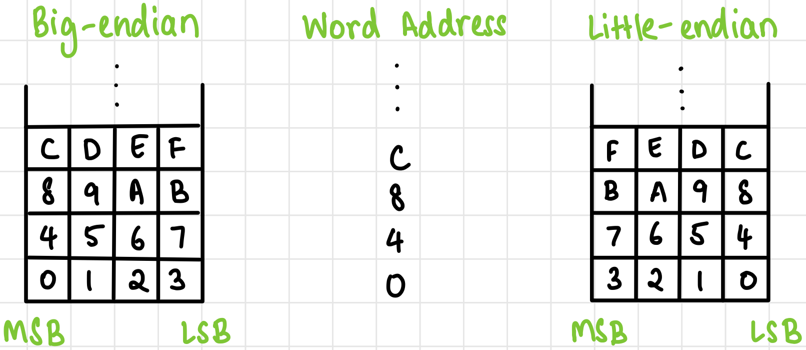
- 0x2C = 44 base 10 = 11th word

Ordering of Bytes

- Big-endian: byte numbers start at the big (most significant) end

- Little-endian: byte numbers that start at the little (least significant) end (x86 uses)

- For 16-bit memory



Big-endian				Word Address	Little-endian			
C	D	E	F	C	F	E	D	C
8	9	A	B	8	B	A	9	8
4	5	6	7	4	7	6	5	4
0	1	2	3	0	3	2	1	0
MSB			LSB		MSB			LSB

- Naming: from Gulliver's Travels

Question 30

Suppose $t0 initially contains 0x23456789. After the following code runs on a big-endian system, what value is $s0?

```
SW  $t0, 0($0)
Lb  $s0, 1($0)
```

big endian: 00000000 $\underset{0}{\underline{2\ 3}}\ \underset{1}{\underline{4\ 5}}\ \underset{2}{\underline{6\ 7}}\ \underset{3}{\underline{8\ 9}}$

$s0 contains 0x00000045

Question 31

Suppose $t0 initially contains 0x23456789. After the following code runs on a little-endian system, what value is $s0?

```
sw $t0, 0($0)
lb $s0, 1($0)
```

little endian: 00000000

23	45	67	89
3	2	1	0

$s0 contains 0x00000067

#4 Good design demands good compromises

- Multiple instruction formats allow flexibility (2 or 3 reg operands)
- Number of instruction formats kept small (adhere to #1 & #3)

3. OPERANDS: CONSTANTS/ IMMEDIATES

- lw and sw use constants or immediates
- Immediately available from instruction
- 16-bit 2's complement number
- addi - add immediate
- subi necessary?

Question 32

C code to MIPS assembly code

```
a = a + 4;
b = a - 12;
```

MIPS Assembly code

```
# $s0 = a, $s1 = b

addi $s0, $s0, 4
addi $s1, $s0, -12
```

constant/ immediate
(4, -12 not stored in registers)

- first has to be register
- immediate can only be second number

Using regular add

```
# $t0 = 4, #$t1 = -12
# $s0 = a, $s1 = b

add $s0, $s0, $t0
add $s1, $s0, $t1
```

Question 33

(a) By writing just assembly language programs, is it possible to determine if the processor running on is Big-Endian or Little-Endian?

(b) What if there are no load byte and store byte instructions?

(a) Yes, by storing a number in a register (4 bytes) and then using load byte instructions to access individual bytes

(refer questions 30, 31 – pages 59, 60)

(b) No, it is not possible

MACHINE LANGUAGE

- computers only understand 1's and 0's (binary representation)
- Assembly language instructions are easy for humans to understand (add, sub etc)
- Each instruction is encoded into 32-bit word (add, sub etc.)
- Three instruction formats
  1. **R-Type**: register operands
  2. **I-Type**: immediate operands
  3. **J-Type**: for jumping

R-TYPE

- 3 register operands
- **rs, rt** : source registers
- **rd**: destination register

Other fields

- **op**: the operation code / opcode (0 for R-type instruction)
- **funct**: the function ; with opcode, tells computer what operation to perform
- **shamt**: the shift amount for shift instructions, otherwise it's 0

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5bits	6 bits

all are 0 for R-type

32-bit word
6 diff bitfields

- MIPS: 32-bit architecture with 32 registers
- Only 5 bits needed to specify register (rs, rt, rd)
- Shift amount: also only 5 bits (beyond which makes no sense for a 32-bit number)
- funct: specifies add/sub
- final machine language code gets fed to microprocessor and executed

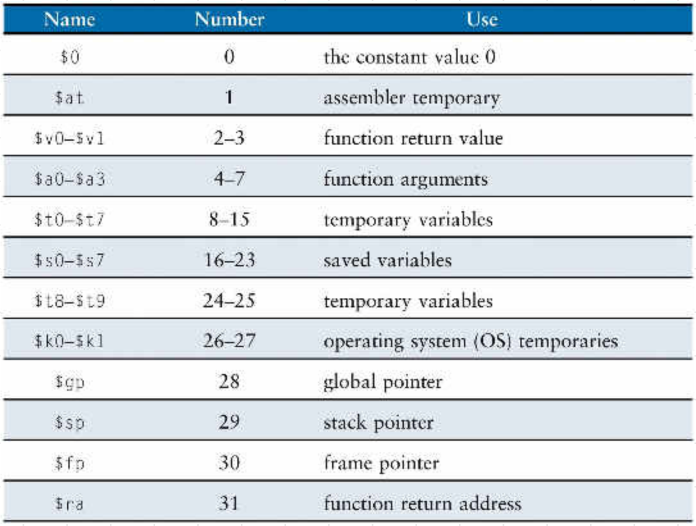
Question 34

Convert assembly code to machine code

```
add $s0, $s1, $s2
sub $t0, $t3, $t5
```

order of registers

table will be given



Name	Number	Use
$0	0	the constant value 0
$at	1	assembler temporary
$v0–$v1	2–3	function return value
$a0–$a3	4–7	function arguments
$t0–$t7	8–15	temporary variables
$s0–$s7	16–23	saved variables
$t8–$t9	24–25	temporary variables
$k0–$k1	26–27	operating system (OS) temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	function return address

Field values

here, source

	op	rs	rt	rd	shamt	funct
add	000000	17	18	16	00000	32
sub	000000	11	13	8	00000	34
	6	5	5	5	5	6

func values

machine language instructions

000000	10001	10010	10000	00000	100000	0x02328020
000000	01011	01101	01000	00000	100010	0x016D4022
6	5	5	5	5	6	

get fed to microprocessor

I-TYPE

- 3 operands
- rs, rt : register operands
- imm: 16-bit two's complement immediate

Other fields

- op: opcode
- all instructions have opcode (simplicity favours regularity)
- operation determined completely by opcode

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

Question 35

Convert assembly code to machine code

```
addi $s0, $s1, 5
addi $t0, $s3, -12
lw $t2, 32($0)
sw $s1, 4($t1)
```

order of registers

```
addi rt, rs, imm
lw rt, imm(rs)
sw rt, imm(rs)
```

acts as destination

op	rs	rt	imm

addi	8	17	16	5
addi	8	19	8	-12
lw	35	0	10	32
sw	43	9	17	4

field values

addi	001000	10001	10000	0000 0000 0000 0101	0x22300005
addi	001000	10011	01000	1111 1111 1111 0100	0x2268FFF4
lw	100011	00000	01010	0000 0000 0010 0000	0x8C0A0020
sw	101011	01001	10001	0000 0000 0000 0100	0xAD310004

↑ machine code

NOTE

- rs is always source
- rd is always destination
- rt can be either source or destination

J-TYPE

- conditional statements / constructs converted to jump / branch instructions (goto)
- 26-bit address operand (addr)
- Used for jump instructions (j)

op	addr
6 bits	26 bits

Summary of Types

R

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

I

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

J

op	addr
6 bits	26 bits

Logical Instructions

combining bit fields

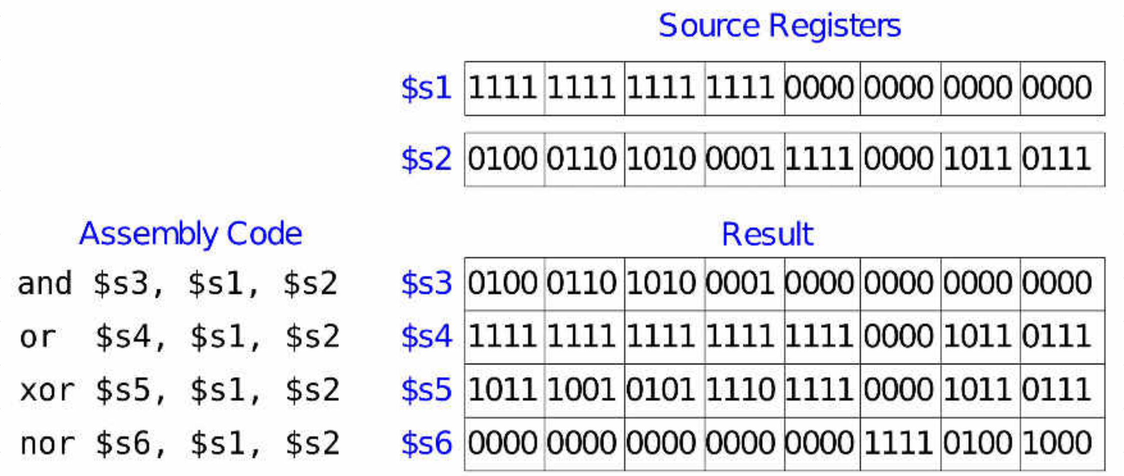
- and, or, xor, nor

masking bits

inverting bits (A NOR $0 = NOT A)

- andi, ori, xori – 16 bit immediate operand is zero extendend (nori not needed)

Logical Instructions

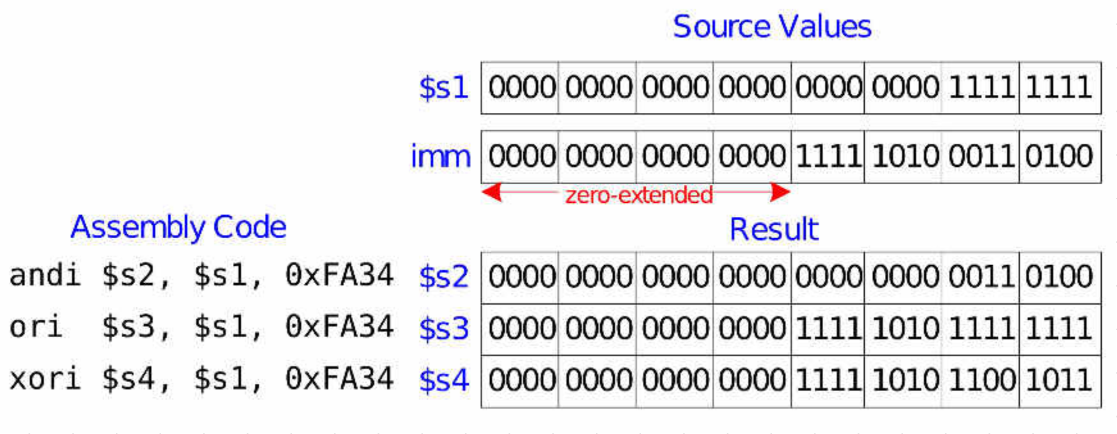


Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code	Result								
and $s3, $s1, $s2	$s3	0100	0110	1010	0001	0000	0000	0000	0000
or $s4, $s1, $s2	$s4	1111	1111	1111	1111	1111	0000	1011	0111
xor $s5, $s1, $s2	$s5	1011	1001	0101	1110	1111	0000	1011	0111
nor $s6, $s1, $s2	$s6	0000	0000	0000	0000	0000	1111	0100	1000

Immediate Operations



Source Values

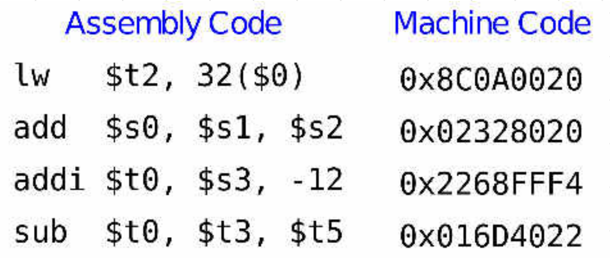
$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

Assembly Code		Result							
andi $s2, $s1, 0xFA34	$s2	0000	0000	0000	0000	0000	0000	0011	0100
ori $s3, $s1, 0xFA34	$s3	0000	0000	0000	0000	1111	1010	1111	1111
xori $s4, $s1, 0xFA34	$s4	0000	0000	0000	0000	1111	1010	1100	1011

Power of the Stored Program

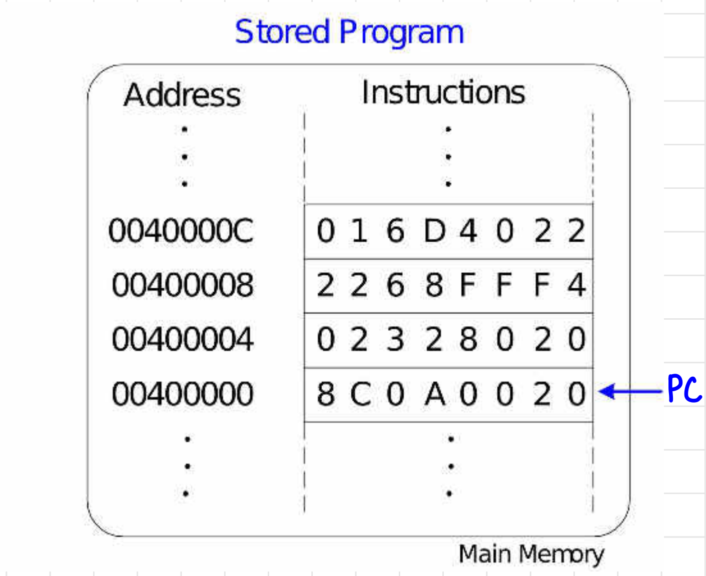
- 32-bit instructions & data stored in memory
- Sequence of instructions: only difference between two applications (word processor, web processor)
- To run new program, no rewiring necessary; only new program to be stored in memory
- Program execution: processor fetches (reads) instructions from memory and performs operation

Stored Program



Assembly Code	Machine Code
`lw $t2, 32($0)`	0x8C0A0020
`add $s0, $s1, $s2`	0x02328020
`addi $t0, $s3, -12`	0x2268FFF4
`sub $t0, $t3, $t5`	0x016D4022

Bottom to Top



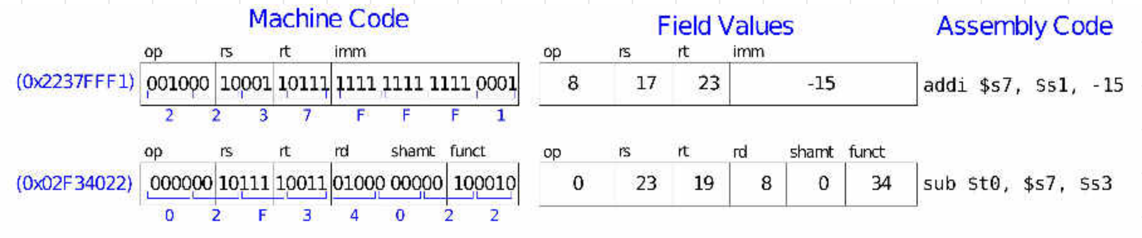
Stored Program

Address	Instructions
⋮	⋮
0040000C	0 1 6 D 4 0 2 2
00400008	2 2 6 8 F F F 4
00400004	0 2 3 2 8 0 2 0
00400000	8 C 0 A 0 0 2 0
⋮	⋮

Interpreting Machine Code

- Start with opcode (tells how to parse)
- If op is all 0's, R-type instruction and function bits tell operation
- Otherwise, opcode tells operation

Machine Code to Assembly



Machine Code

	op	rs	rt	imm			
(0x2237FFF1)	001000	10001	10111	1111 1111 1111 0001			
	2 2		3 7	F F F 1			

	op	rs	rt	rd	shamt	funct
(0x02F34022)	000000	10111	10011	01000	00000	100010
	0 2		F 3	4 0		2 2

Field Values / **Assembly Code**

op	rs	rt	imm			Assembly Code
8	17	23	-15			addi $s7, $s1, -15

op	rs	rt	rd	shamt	funct	Assembly Code
0	23	19	8	0	34	sub $t0, $s7, $s3

© vibha's notes 2020

Braching Instructions

- Execute instructions out of sequence
- Types of branches
  1) Conditional
     - branch if equal (beq)
     - branch if not equal (bne)
  2) Unconditional
     - jump (j)
     - jump register (jr)
     - jump and link (jal)

Conditional Branching

- beq

like goto statement

branch if equal

```
addi $s0, $0, 4            # $s0 = 0+4 =4
addi $s1, $0, 2            # $s1 = 0+2 =2
add $s1, $s1, $s1          # $s1 = 2+2 =4
beq $s0, $s1, target       # branch is taken
addi $s1, $s1, 1           # not executed  ]
sub $s1, $s1, $s0          # not executed  ] skips

target:                    # label
 add $s1, $s1, $s0         # $s1 = 4+4 = 8
```

- if 2 registers ($s0 & $s1) are equal, target branch is taken

- bne

```
addi $s0, $0, 4           # $s0 = 0+4 = 4
addi $s1, $0, 2           # $s1 = 0+2 = 2
add $s1, $s1, $s1         # $s1 = 2+2 = 4
bne $s0, $s1, target      # branch not taken
addi $s1, $s1, 1          # executed
sub $s1, $s1, $s0         # executed

target:                   # label
 add $s1, $s1, $s0        # executed
```

Unconditional Branching

- j

```
addi $s0, $0, 4           # $s0 = 4
addi $s1, $0, 1           # $s1 = 1
j target                  # jump to target
sra $s1, $s1, 2           # not executed
addi $s1, $s1, 1          # not executed
sub $s1, $s1, $s0         # not executed

target:                   # label
 add $s1, $s1, $s0        # $s1 = 5
```

sra: shift right arithmatic (sign-extended)

srl: 0-extended

- jr

```
0x00002000    addi $s0, $0, 0x2010
0x00002004    jr $s0
0x00002008    addi $s1, $0, 1
0x0000200C    sra $s1, $s1, 2
0x00002010    lw $s3, 44($s1)
```

- jr is an R-type instruction

Question 36

Convert to MIPS assembly code (IF-statement)

```
if (i == j)
    f = g + h;

f = f - i;
```

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j

bne $s3, $s4, L1
add $s0, $s1, $s2

L1: sub $s0, $s0, $s3
```

Assembly tests opposite case of high level code (==, bne)

Question 37

```
if (i == j)        (IF-ELSE block)
    f = g + h;

else
    f = f - i;
```

```
# $s0 = f, $s1 = g, $s2 = h
# $s3 = i, $s4 = j

bne $s3, $s4, L1
add $s0, $s1, $s2
j done     # to skip else (preserve absolute
                   mutual exclusivity)

L1: sub $s0, $s0, $s3
done:
```

Question 38

```
// determines the power        (WHILE)
// of x such that 2^x = 128

int pow = 1;
int x = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

```
# $s0 = pow, $s1 = x

       addi $s0, $0, 1       # pow=1
       add $s1, $0, $0       #  x=0
       addi $t0, $0, 128     # temp=128
while: beq $s0, $t0, done    # if pow=128, break
       sll $s0, $s0, 1       # shift left (x2)
       addi $s1, $s1, 1      # add 1
       j while               # jump to start of loop
done:
```

shift left logical ← sll

Question 39 (FOR)

```
// add numbers from 0 to 9

int sum = 0;
int i;

for (i = 0; i != 10; ++i) {
    sum = sum + i;
}
```

```
# $s0 = i, $s1 = sum
     add $s1, $0, $0
     add $s0, $0, $0
     add $t0, $0, 10
for: beq $s0, $t0, done
     add $s1, $s1, $s0
     addi $s0, $s0, 1
     j for
done:
```

Question 40

Assume that \$s1 contains 7 and \$s2 contains 4. Consider the following instruction sequence (the nor instruction inverts each bit). What value gets stored in \$s3? What operation is being performed?

```
nor $s2, $s0, $s2     # s2 = not 4
addi $s2, $s2, 1      # s2 = 2's comp(4) = -4
add $s3, $s1 ,$s2     # s3 = 7-4 = 3
```

2's complement subtraction

\$s3 contains 3