Number Representation and Computer Arithmetic

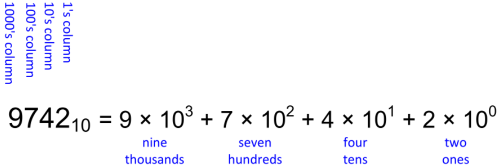
University of South Carolina

Introduction to Computer Architecture
Fall, 2024
Mehdi Yaghouti

Molinaroli College of Engineering and Computing
UNIVERSITY OF SOUTH CAROLINA

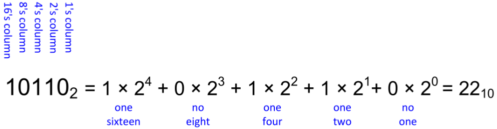
Binary Number Representation

- **Decimal system**:
  - Base: 10
  - Digits: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - Representation of a decimal number:



$$9742_{10} = \underset{\substack{\text{nine}\\\text{thousands}}}{9 \times 10^3} + \underset{\substack{\text{seven}\\\text{hundreds}}}{7 \times 10^2} + \underset{\substack{\text{four}\\\text{tens}}}{4 \times 10^1} + \underset{\substack{\text{two}\\\text{ones}}}{2 \times 10^0}$$

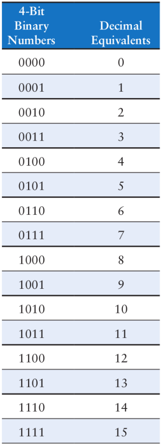
- **Binary system**:
  - Base: 2
  - Digits: $\{0, 1\}$
  - Representation of a binary number:



$$10110_2 = \underset{\substack{\text{one}\\\text{sixteen}}}{1 \times 2^4} + \underset{\substack{\text{no}\\\text{eight}}}{0 \times 2^3} + \underset{\substack{\text{one}\\\text{four}}}{1 \times 2^2} + \underset{\substack{\text{one}\\\text{two}}}{1 \times 2^1} + \underset{\substack{\text{no}\\\text{one}}}{0 \times 2^0} = 22_{10}$$

Unsigned Binary Numbers

- N-bit binary number can represent 2^N numbers,
  - Minimum: $\overbrace{0\ldots0}^{\text{N-bits}}$
  - Maximum: $\overbrace{1\ldots1}^{\text{N-bits}}$
  - Range: $\left[0,\ldots,2^N-1\right]$
- Decimal to binary conversion
  - Using common powers of 2
    $1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096$
  - Repeated Divisions
- Binary to decimal conversion
  $(b_N\ldots b_0)_2 = \sum_{k=0}^{N} 2^k b_k$



4-Bit Binary Numbers	Decimal Equivalents
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Hexadecimal Representation

- **Hexadecimal system**:
  - Base: 16
  - Digits: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$
  - Representation of a hexadecimal number:

256's column
16's column
1's column

$$2ED_{16} = \underset{\substack{\text{two}\\\text{two hundred fifty six's}}}{2 \times 16^2} + \underset{\substack{\text{fourteen}\\\text{sixteens}}}{E \times 16^1} + \underset{\substack{\text{thirteen}\\\text{ones}}}{D \times 16^0} = 749_{10}$$

- Correspondence between Hex digits and 4-bits $\sum_{k=0}^{N} 2^k b_k = \sum_{k=0}^{N/4} 16^k \left(\sum_{l=0}^{3} 2^l b_{(4*k+l)}\right)$
- Binary to Hexadecimal conversion
  - Pack each 4-bit into a hex digit
- Hexadecimal to Binary conversion
  - Unpack each hex digits into 4-bits

DEAFDAD8 = 1101 1110 1010 1111 1101 1010 1101 1000

Hexadecimal Digit	Binary Equivalent
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Octal Representation

- **Octal system**:
  - Base: 8
  - Digits: $\{0, 1, 2, 3, 4, 5, 6, 7\}$
  - Representation of an octal number:

64's column
8's column
1's column

$$367_8 = \underset{\text{three sixty fours}}{3\times 8^2} + \underset{\text{six eights}}{6\times 8^1} + \underset{\text{seven ones}}{7\times 8^0} = 247_{10}$$

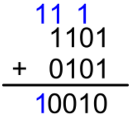
- Correspondence between Octal digits and 3-bits
  $\sum_{k=0}^{N} 2^k b_k = \sum_{k=0}^{N/3} 8^k \left(\sum_{l=0}^{3} 2^l b_{(3*k+l)}\right)$
- Binary to Octal conversion
  - Pack each 3-bit into a octal digit
- Octal to Binary conversion
  - Unpack each octal digits into 3-bits

$$(2357)_8 = 010\ \ 011\ \ 101\ \ 111$$

Octal Digit	Binary Equivalent
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Binary Addition

- **Basic Rules**:
  - $0 + 0 = 0$
  - $0 + 1 = 1$
  - $1 + 0 = 1$
  - $1 + 1 = 10$ (0 with a carry of 1)
- **Steps for Adding Two Binary Numbers**:
  1. Aligning the Numbers
  2. Adding digit by digit, starting from LSB
  3. Carry Over: If the sum is 2 (binary 10), carry the 1 to the next column
  4. Overflow: The carry out of the leftmost digit

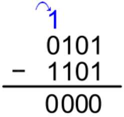


$$\begin{array}{r} 11\ 1 \\ 1101 \\ +\ \ 0101 \\ \hline 10010 \end{array}$$

- Care must be taken as registers have constant number of bits

Binary Subtraction

- **Basic Rules**:
  - $0 - 0 = 0$
  - $1 - 0 = 1$
  - $1 - 1 = 0$
  - $0 - 1 = 1$ (with a borrow of 1 from the next higher bit)
- **Steps for Subtracting Two Binary Numbers**:
  1. Aligning the numbers
  2. Subtracting digit by digit, starting from LSB
  3. Borrowing: When subtracting 1 from 0, borrow 1 from the next higher bit
  4. Borrow in: If a borrow needed beyond the leftmost digit, the result is negative



Sign/Magnitude Number System

- **Sign/Magnitude**:
  - Sign bit: Most significant bit

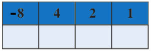
$$\begin{cases} + \to 0 \\ - \to 1 \end{cases}$$

Example:

$$\begin{aligned} +23 &= 00010111 \\ -23 &= 10010111 \qquad (\mathit{sign/magnitude\ representation}) \end{aligned}$$

  - Spans the range $\left[-\left(2^{N-1}-1\right), 2^{N-1}-1\right]$
  - Ordinary addition doesn't work on sign included representation
  - Zero has two representations $+0$, -0
  - Troublesome to use in fast arithmetic circuits

Two's Complement Number System



-8	4	2	1

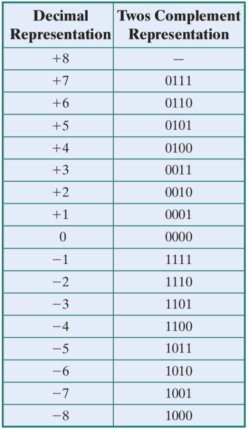
- **Two's Complement system**:
  - N-bits Two's complement representation

$$(b_{N-1} \dots b_0)_2 = -b_N 2^{N-1} + \sum_{k=0}^{N-2} 2^k b_k$$

  - Spans the asymmetric range $\left[-2^{N-1}, 2^{N-1}-1\right]$
  - **Ordinary addition works well on sign included representation**
  - Sign bit: Most significant bit

$$\begin{cases} + \rightarrow 0 \\ - \rightarrow 1 \end{cases}$$

  - Zero has only one representation
  - Two's complement is the most accepted one for fast arithmetic circuits



Decimal Representation	Twos Complement Representation
+8	—
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	0000
−1	1111
−2	1110
−3	1101
−4	1100
−5	1011
−6	1010
−7	1001
−8	1000

Two's Complement System

- **Negation Rule**
  1. Negate all the bits
  2. Add 1

  Example:

$$+23 = 00010111 \xrightarrow{\text{negated}} 11101000 \xrightarrow{+1} 11101001 = -23$$

$$-23 = 11101001 \xrightarrow{\text{negated}} 00010110 \xrightarrow{+1} 00010111 = +23$$

- **Addition/Subtraction**

$47 + 11 = 58$

$$\begin{array}{r} 00101111 \\ +00001011 \\ \hline 00111010 \end{array}$$

$47 - 11 = 36$

$$\begin{array}{r} 00101111 \\ +\ \ 11110101 \\ \hline 100100100 \end{array}$$

$-47 + 11 = -36$

$$\begin{array}{r} 11010001 \\ +00001011 \\ \hline 11011100 \end{array}$$

$-47 - 11 = -58$

$$\begin{array}{r} 11010001 \\ +\ \ 11110101 \\ \hline 111000110 \end{array}$$

Overflow

- **Unsigned Numbers**
  - Overflow occurs when there is a carry out of the MSB column

Example:

$$7 + 12 = 19$$

$$\begin{array}{r} 0111 \\ +\ 1100 \\ \hline 11011 \end{array}$$

- **Signed Numbers (Two's complement)**
  - Overflow can only happen when two numbers have the same sign bit
  - Overflow occurs when the result has the opposite sign bit

Example:

$$7 + 4 = 11$$

$$\begin{array}{r} 0111 \\ +0100 \\ \hline 1011 \end{array}$$

$$-7 - 4 = -11$$

$$\begin{array}{r} 1001 \\ +\ 1100 \\ \hline 10101 \end{array}$$

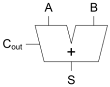
Adder

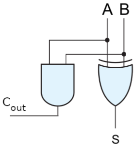
- **Half Adder:** $\{c_{out}, S\} = A + B$

Half Adder

A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$
$C_{out} = AB$



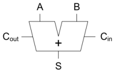


Adder

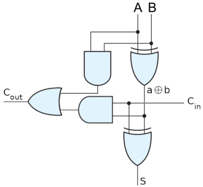
- **Half Adder:** $\{c_{out}, S\} = A + B$
- **Full Adder:** $\{c_{out}, S\} = A + B + c_{in}$
- **Generate:** $G = A \bullet B$
- **Propagate:** $P = A \oplus B$

Full Adder

C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



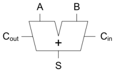
$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$



Adder

- **Half Adder:** $\{c_{out}, S\} = A + B$
- **Full Adder:** $\{c_{out}, S\} = A + B + c_{in}$
- **Generate:** $G = A \bullet B$
- **Propagate:** $P = A \oplus B$
- **Delay:**
  $max\{2\, t_{pd\_xor}, t_{pd\_xor} + t_{pd\_or} + t_{pd\_and}\}$

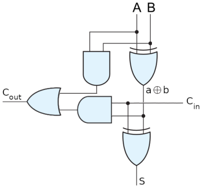
Full Adder



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + AC_{in} + BC_{in}$



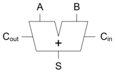
Adder

- **Half Adder:** $\{c_{out}, S\} = A + B$
- **Full Adder:** $\{c_{out}, S\} = A + B + c_{in}$
- **Generate:** $G = A \bullet B$
- **Propagate:** $P = A \oplus B$
- **Delay:** $max\{2\,t_{pd\_xor}, t_{pd\_xor} + t_{pd\_or} + t_{pd\_and}\}$

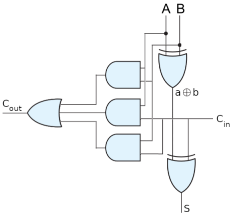
- $t_{FA} = t_{pd\_or3} + t_{pd\_and}$

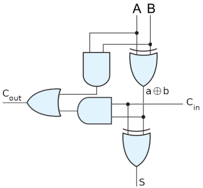
Full Adder

C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



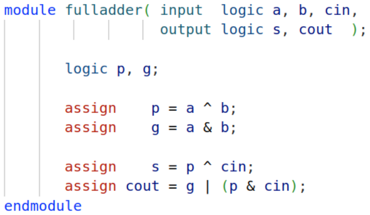
$S = A \oplus B \oplus C_{in}$
$C_{out} = AB + AC_{in} + BC_{in}$



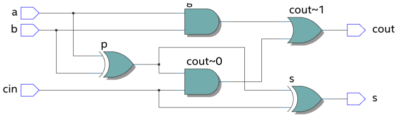


SystemVerilog (Optional)

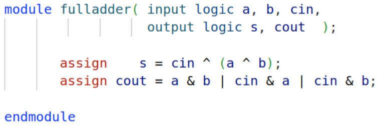
- Implementation using P and G signals



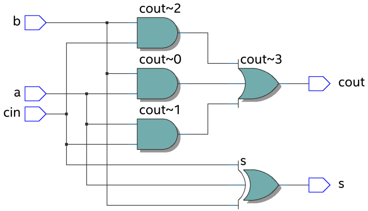
```
module fulladder( input  logic a, b, cin,
                  output logic s, cout  );

      logic p, g;

      assign    p = a ^ b;
      assign    g = a & b;

      assign    s = p ^ cin;
      assign cout = g | (p & cin);
endmodule
```



- The second alternative



```
module fulladder( input logic a, b, cin,
                  output logic s, cout  );

      assign    s = cin ^ (a ^ b);
      assign cout = a & b | cin & a | cin & b;

endmodule
```

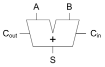


Ripple-Carry Adder

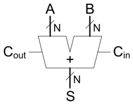
- **Carry Propagate Adder (CPA):** Sums N-bit inputs
- **Ripple-Carry Adder:** Chains N full adders
- **Delay:** $t_{pd\_rca} = N_t \, t_{FA}$

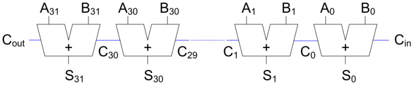
Full Adder

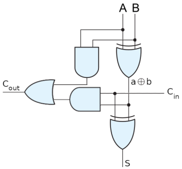
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



$S = A \oplus B \oplus C_{in}$
$C_{out} = AB + AC_{in} + BC_{in}$

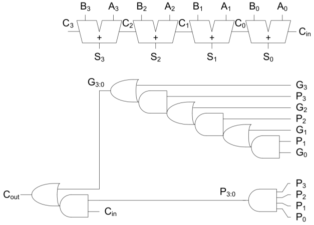


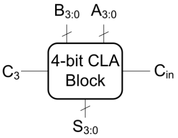


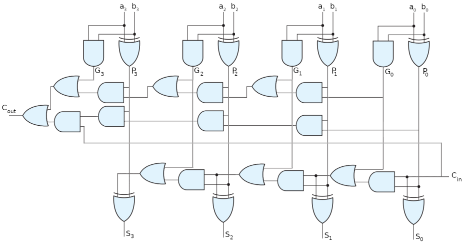


Look-ahead Block

- **Generate:**
  $G_{j:i} = G_j + P_j \, G_{j-1:i}$
- **Propagate:**
  $P_{j:i} = P_j \, P_{j-1} \dots P_i$
- **Carry:**
  $C_{out} = G_{j:i} + P_{j:i} C_{in}$
- **Delay:**
  $t_{pg} = max\{t_{AND}, t_{XOR}\}$
  $t_{pg\_block} = 3\ (t_{AND} + t_{OR})$

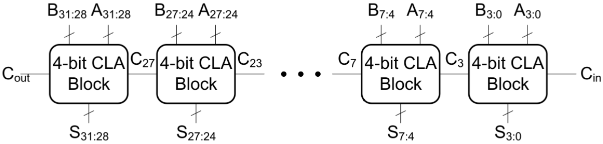






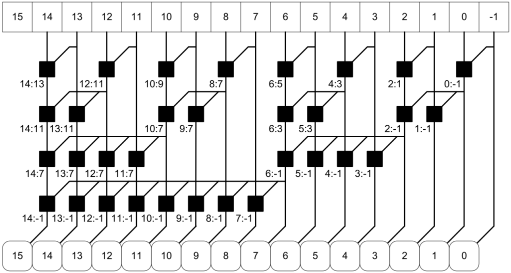
Carry Look-Ahead Adder

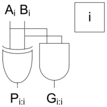
- **Generate:** $G_{j:i} = G_j + P_j\, G_{j-1:i}$
- **Propagate:** $P_{j:i} = P_j\, P_{j-1} \ldots P_i$
- **Delay:** $t_{cla} = t_{pg} + t_{pg\_block} + \left(\frac{N}{k} - 1\right) t_{and\_or} + k\, t_{FA}$

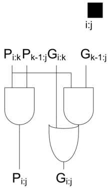


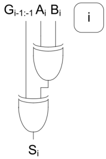
Prefix Adder (Optional)

- **Generate:** $G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$
- **Propagate:** $P_{i:j} = P_{i:k}\, P_{k-1:j}$
- **Delay:** $t_{PA} = t_{pg} + t_{pg\_prefix}\; log_2 N + t_{XOR}$



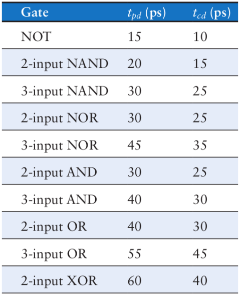






Question

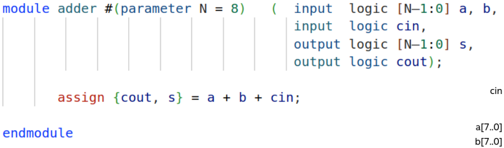
- Compare the delays of a 64-bit ripple-carry, a 64-bit carry-lookahead with 4-bit blocks and a 64-bit prefix adder.



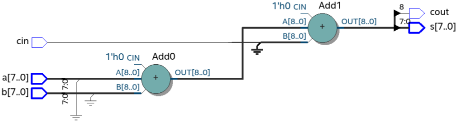
Gate	t_{pd} (ps)	t_{cd} (ps)
NOT	15	10
2-input NAND	20	15
3-input NAND	30	25
2-input NOR	30	25
3-input NOR	45	35
2-input AND	30	25
3-input AND	40	30
2-input OR	40	30
3-input OR	55	45
2-input XOR	60	40

SystemVerilog (Optional)

- keyword `parameter`
- Using high level description we leave the implementation to the synthesizer

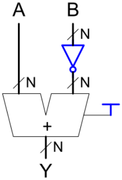


```
module adder #(parameter N = 8)   (  input  logic [N-1:0] a, b,
                                     input  logic cin,
                                     output logic [N-1:0] s,
                                     output logic cout);

         assign {cout, s} = a + b + cin;

endmodule
```



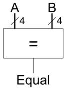
Subtraction

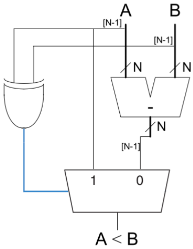
- **Two's Complement**
- **Subtraction:** $A - B = A + \overline{B} + 1$

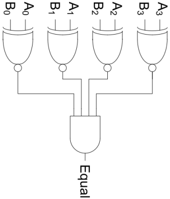


Comparators

- Compares two binary inputs
- Using XNOR to check equality
- Using subtraction and check the sign bit







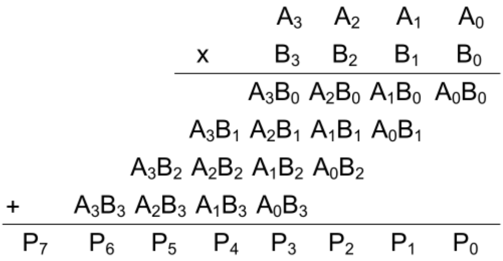
Multiplication

- Binary multiplication is based on two basic operations:
    - Generation of partial products
    - Accumulation
- The multiplication of two N-bits number is in general a $2N$-bits number

$$
\begin{array}{ccccccccc}
 & & & & & 1 & 1 & 0 & 1 \\
 & & & & \times & 1 & 1 & 1 & 0 \\
\cline{5-9}
 & & & & & 0 & 0 & 0 & 0 \\
 & & & & 1 & 1 & 0 & 1 & \\
 & & & 1 & 1 & 0 & 1 & & \\
+ & & 1 & 1 & 0 & 1 & & & \\
\hline
 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0
\end{array}
$$

Multiplication

- Binary multiplication is based on two basic operations:
  - Generation of partial products
  - Accumulation
- The multiplication of two N-bits number is a 2N-bits number in general
- Each partial product is either zero or the multiplicand

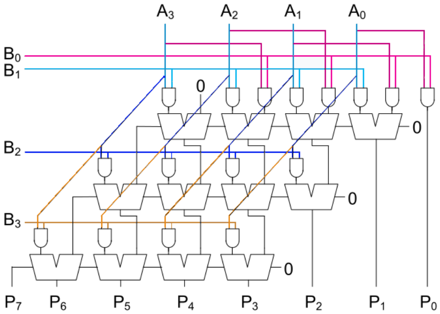


					A_3	A_2	A_1	A_0
x					B_3	B_2	B_1	B_0
					A_3B_0	A_2B_0	A_1B_0	A_0B_0
				A_3B_1	A_2B_1	A_1B_1	A_0B_1	
			A_3B_2	A_2B_2	A_1B_2	A_0B_2		
+		A_3B_3	A_2B_3	A_1B_3	A_0B_3			
	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

Array Multiplier

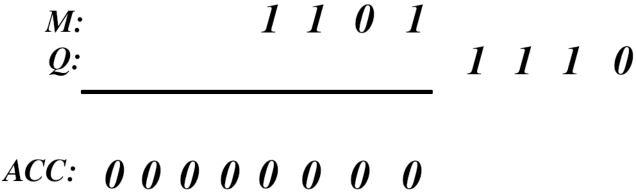
- Binary multiplication is based on two basic operations:
  - Generation of partial products
  - Accumulation
- The multiplication of two N-bits number is a 2N-bits number in general
- Each partial product is either zero or the multiplicand
- Partial products can be generated by using AND gates

$$\begin{array}{cccccccc}
 & & & & A_3 & A_2 & A_1 & A_0 \\
 & & & x & B_3 & B_2 & B_1 & B_0 \\ \hline
 & & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 & & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & \\
 & & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\ \hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}$$



Sequential Multiplier (Optional)

- Binary multiplication can be done in a sequential manner
- Sequential multiplication is based on two observations
  - Each partial product is either zero of the multiplicand
  - The partial products can be accumulated incrementally
- It can be best understood by an example



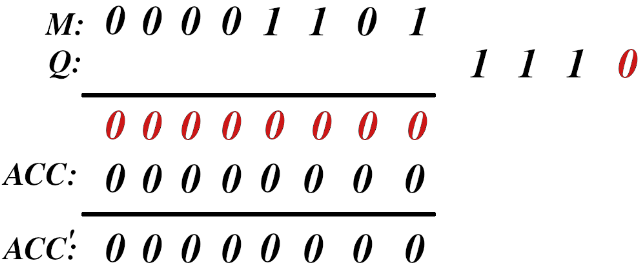
M: 1 1 0 1

Q: 1 1 1 0

ACC: 0 0 0 0 0 0 0 0

Sequential Multiplier (Optional)

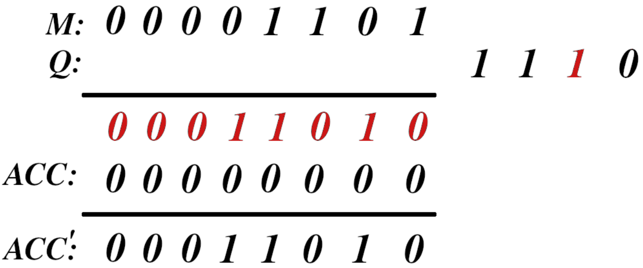
- Binary multiplication can be done in a sequential manner
- Sequential multiplication is based on two observations
  - Each partial product is either zero of the multiplicand
  - The partial products can be accumulated incrementally
- It can be best understood by an example



```
M:    0 0 0 0 1 1 0 1
Q:                      1 1 1 0
      _______________
      0 0 0 0 0 0 0 0
ACC:  0 0 0 0 0 0 0 0
      _______________
ACC': 0 0 0 0 0 0 0 0
```

Sequential Multiplier (Optional)

- Binary multiplication can be done in a sequential manner
- Sequential multiplication is based on two observations
  - Each partial product is either zero of the multiplicand
  - The partial products can be accumulated incrementally
- It can be best understood by an example



M: 0 0 0 0 1 1 0 1

Q: 1 1 1 0

0 0 0 1 1 0 1 0

ACC: 0 0 0 0 0 0 0 0

ACC': 0 0 0 1 1 0 1 0

Sequential Multiplier (Optional)

- Binary multiplication can be done in a sequential manner
- Sequential multiplication is based on two observations
    - Each partial product is either zero of the multiplicand
    - The partial products can be accumulated incrementally
- It can be best understood by an example

M:	0	0	0	0	1	1	0	1				
Q:									1	1	1	0
	0	0	1	1	0	1	0	0				
ACC:	0	0	0	1	1	0	1	0				
ACC':	0	1	0	0	1	1	1	0				

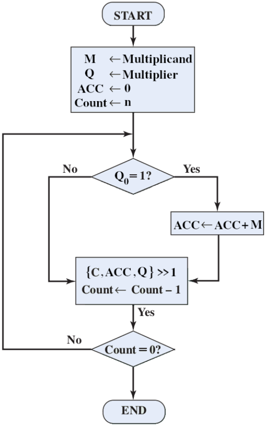
Sequential Multiplier (Optional)

- Binary multiplication can be done in a sequential manner
- Sequential multiplication is based on two observations
  - Each partial product is either zero of the multiplicand
  - The partial products can be accumulated incrementally
- It can be best understood by an example

M:	0	0	0	0	1	1	0	1				
Q:									1	1	1	0
	0	1	1	0	1	0	0	0				
ACC:	0	1	0	0	1	1	1	0				
ACC':	1	0	1	1	0	1	1	0				

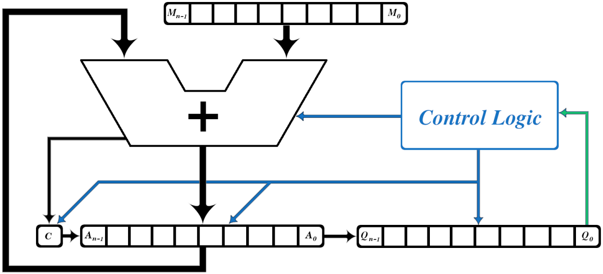
Sequential Multiplier (Optional)

- Binary multiplication can be done in a sequential manner
- Sequential multiplication is based on two observations
  - Each partial product is either zero of the multiplicand
  - The partial products can be accumulated incrementally
- The process can be described with the following flowchart



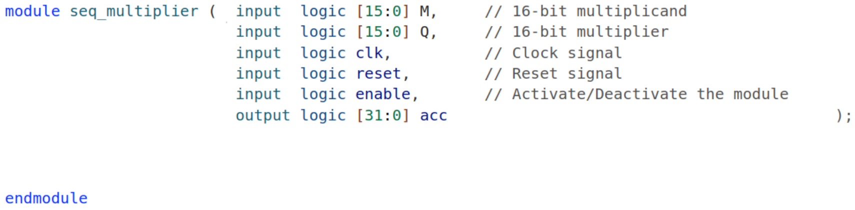
Sequential Multiplier (Optional)

- Hardware Architecure
  - M holds the multplicand
  - Q holds the multiplier
  - A holds the partial products summation



SystemVerilog Project: Sequential Multiplier (Optional)

- Design a system verilog module to perform the sequential multiplication
- Your module must have the following interface

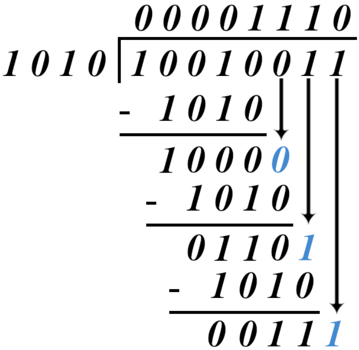


```
module seq_multiplier (  input  logic [15:0] M,     // 16-bit multiplicand
                         input  logic [15:0] Q,     // 16-bit multiplier
                         input  logic clk,          // Clock signal
                         input  logic reset,        // Reset signal
                         input  logic enable,       // Activate/Deactivate the module
                         output logic [31:0] acc                                              );



endmodule
```

- The project is optional and has extra bonus
- The following files must be uploaded
    - The multiplier module: *Seq_Multiplier.sv*
    - Testbench: *Testbench.sv*
    - Simulated Waveforms: *Waveforms.pdf*
    - Comparing the sequential and array multiplier regarding the delay time and resource requirements

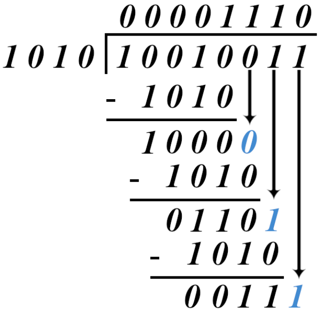
Division (Optional)

- Binary division is based on two basic operations:
    - Generation of partial remainders
    - Subtraction and shifting
- The quotient in division of two N-bits number is in general a N-bits number



Division (Optional)

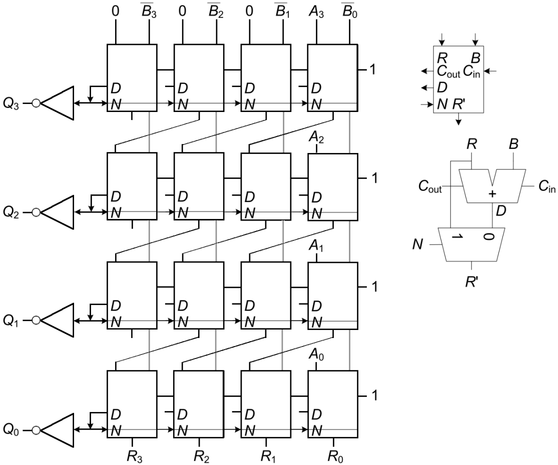
- Binary division is based on two basic operations:
  - Generation of partial remainders
  - Subtraction and shifting
- The quotient in division of two N-bits number is in general a N-bits number



```
$R' \leftarrow 0$
for $i = (N-1)$ $to$ $0$
    $R \leftarrow \{R' << 1, A_i\}$
    $D = R - B$
    $if$ $D < 0$
        $Q_i \leftarrow 0$
        $R' \leftarrow R$
    $else$
        $Q_i \leftarrow 1$
        $R' \leftarrow D$
$R \leftarrow R'$
```

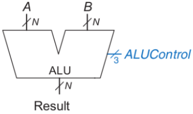
Array Divider (Optional)

- Hardware implementation of binary division



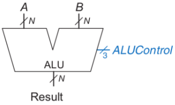
ALU

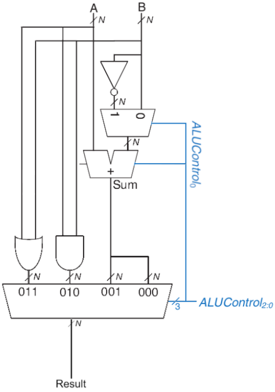
- Performs various mathematical and logical operations
- The desired result can be selected by `ALUControl`



ALU

- Performs various mathematical and logical operations
- The desired result can be selected by `ALUControl`
- An ALU performing `ADD`, `SUB`, `AND` and `OR`

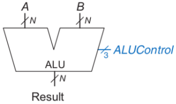




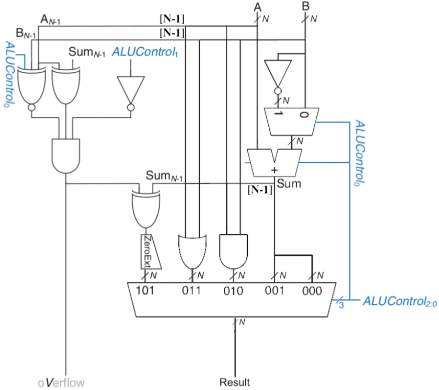
$ALUControl_{2:0}$	Function
000	Add
001	Subtract
010	AND
011	OR

ALU

- Performs various mathematical and logical operations
- The desired result can be selected by `ALUControl`
- An ALU performing `ADD`, `SUB`, `AND`, `OR` and `SLT`

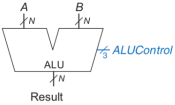


ALUControl$_{2:0}$	Function
000	Add
001	Subtract
010	AND
011	OR
101	SLT

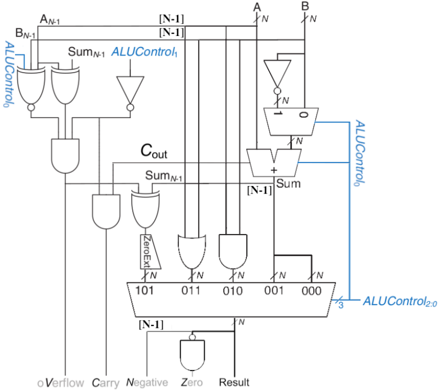


ALU

- Performs various mathematical and logical operations
- The desired result can be selected by `ALUControl`
- An ALU performing `ADD`, `SUB`, `AND`, `OR` and `SLT`
- Common flags: `Negative`, `Zero`, `Carry` and `oVerflow`

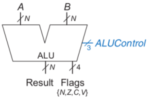


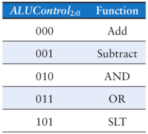
$ALUControl_{2:0}$	Function
000	Add
001	Subtract
010	AND
011	OR
101	SLT



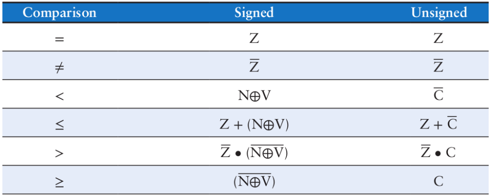
ALU

- Performs mathematical and logical operations
- ALUControl specifies the function
- Common Flags: `N`, `Z`, `C`, `V`
- Overflow detection
- Comparison depends on signed/unsigned

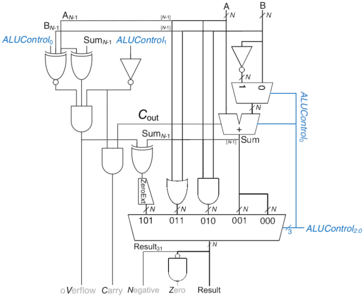




$ALUControl_{2:0}$	Function
000	Add
001	Subtract
010	AND
011	OR
101	SLT

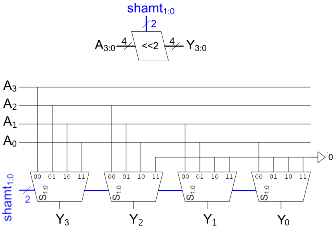


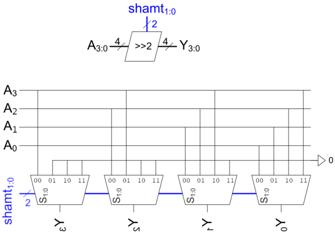
Comparison	Signed	Unsigned
=	Z	Z
≠	$\overline{Z}$	$\overline{Z}$
<	$N\oplus V$	$\overline{C}$
≤	$Z + (N\oplus V)$	$Z + \overline{C}$
>	$\overline{Z} \bullet (\overline{N\oplus V})$	$\overline{Z} \bullet C$
≥	$(\overline{N\oplus V})$	C



Shifters

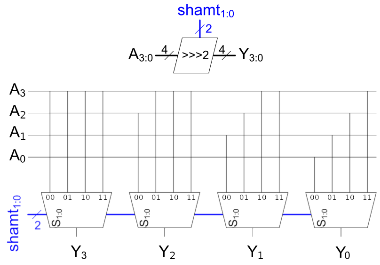
- Logical left and right shifts





Shifters

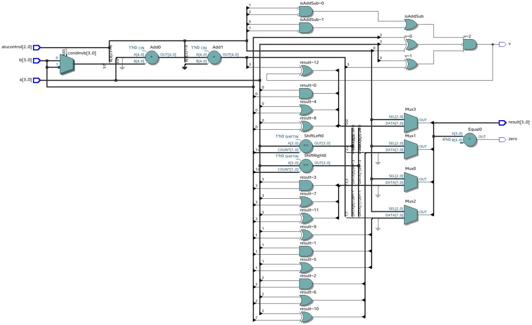
- Logical shifter
- Arithmetic shifter
- Arithmetic shift left multiplies by 2
- Arithmetic shift right divides by 2
- Overflow must be taken care of
- N-bit shifter can be built from N, $N:1$ MUXs



SystemVerilog (Optional)

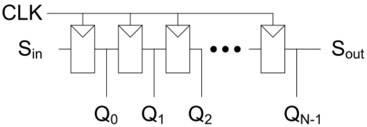
- A typical ALU supporting `add`, `sub`, `and`, `or`, `slt`, `sll`, `srl`
- It has generates `Zero` and `oVerflow` signals

```
module ALU #(parameter N=4, M=3)
      (input  logic [N-1:0] a, b,
       input  logic [M-1:0]  alucontrol,
       output logic [N-1:0] result,
       output logic  zero, v);
  logic [N:0] condinvb, sum;
  logic        isAddSub;

  assign condinvb = alucontrol[0] ? ~b : b;
  assign sum = a + condinvb + alucontrol[0];
  assign isAddSub = ~alucontrol[2] & ~alucontrol[1] |
                    ~alucontrol[1] & alucontrol[0];
  always_comb
    case (alucontrol)
      3'b000:  result = sum;          // add
      3'b001:  result = sum;          // subtract
      3'b010:  result = a & b;        // and
      3'b011:  result = a | b;        // or
      3'b100:  result = a ^ b;        // xor
      3'b101:  result = sum[N-1] ^ v; // slt
      3'b110:  result = a << b[1:0];  // sll
      3'b111:  result = a >> b[1:0];  // srl
      default: result = 4'bx;
    endcase

  assign zero = (result == 32'b0);
  assign v = ~(alucontrol[0] ^ a[31] ^ b[31]) & (a[31] ^ sum[31]) & isAddSub;

endmodule
```



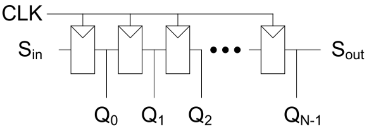
Serial-to-Parallel/Parallel-to-Serial Converters

- Shift registers act as serial-to-parallel converters
- The input is provided serially at S_{in}
- After N cycles, the last N inputs appear at $Q_0 \ldots Q_{N-1}$

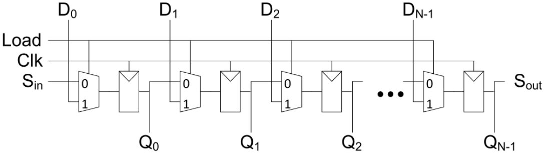


Serial-to-Parallel/Parallel-to-Serial Converters

- Shift registers act as serial-to-parallel converters
- The input is provided serially at S_{in}
- After N cycles, the last N inputs appear at $Q_0 \ldots Q_{N-1}$

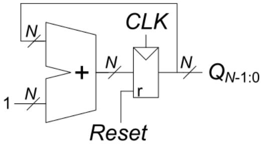


- Shift registers with parallel load can act in reverse
- The input $D_0 \ldots D_{N-1}$ is loaded in parallel
- Takes N cycles to shift out as parallel-to-serial converter



Counters

- N-bit counter composed of an adder and a resettable register
- There different approaches for designing various counters
- One easy way is to use an adder
- On each cycle, the counter adds 1 to the value stored in the register
- They are commonly used for dividing the clock frequency


N
1
N
+
N
CLK
r
Reset
N
Q_{N-1:0}