Computer Arithmetic

Readings: 3.1-3.3, A.5

Review binary numbers, 2's complement

Develop Arithmetic Logic Units (ALUs) to perform CPU functions.

Introduce shifters, multipliers, etc.

Binary Numbers

Decimal: $469 = 4*10^2 + 6*10^1 + 9*10^0$

Binary: $01101 = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = (13)_{10}$

Example: $0111010101 = (?)_{10}$

2's Complement Numbers

Positive numbers & zero have leading 0, negative have leading 1

Negation: Flip all bits and add 1

Ex: $-(01101)_2 =$

To interpret numbers, convert to positive version, then convert:

11010 =

01100 =

Sign Extension

Conversion of n-bit to (n+m)-bit 2's complement: replicate the sign bit

$b_3b_2b_1b_0 = b_3b_3b_3b_2b_1b_0 = b_3b_3b_3b_3b_3b_3b_3b_3b_3b_3b_3b_3b_3b_2b_1b_0$

Ex - Convert to 8-bit: $01101 = (13)_{10}$ $11101 = (-3)_{10}$

Arithmetic Operations

Decimal:

```
   5 7 8 9 2
 + 7 8 9 5 6
 -----------
```

Binary:

```
   1 0 1 0 1 1 1
 + 0 1 0 0 1 0 1
 ---------------
```

Binary:

```
   1 0 1 0 0 1 1 0          1 0 1 0 0 1 1 0
 - 0 0 0 1 0 1 1 1   ==>  + _______________
 -----------------
```

Overflows

Operations can create a number too large for the number of bits

n-bit 2's complement can hold $-2^{(n-1)} \ldots 2^{(n-1)}-1$

Can detect overflow in addition when highest bit has carry-in ≠ carry-out

(carry-in) ⊕ (carry-out) = 1

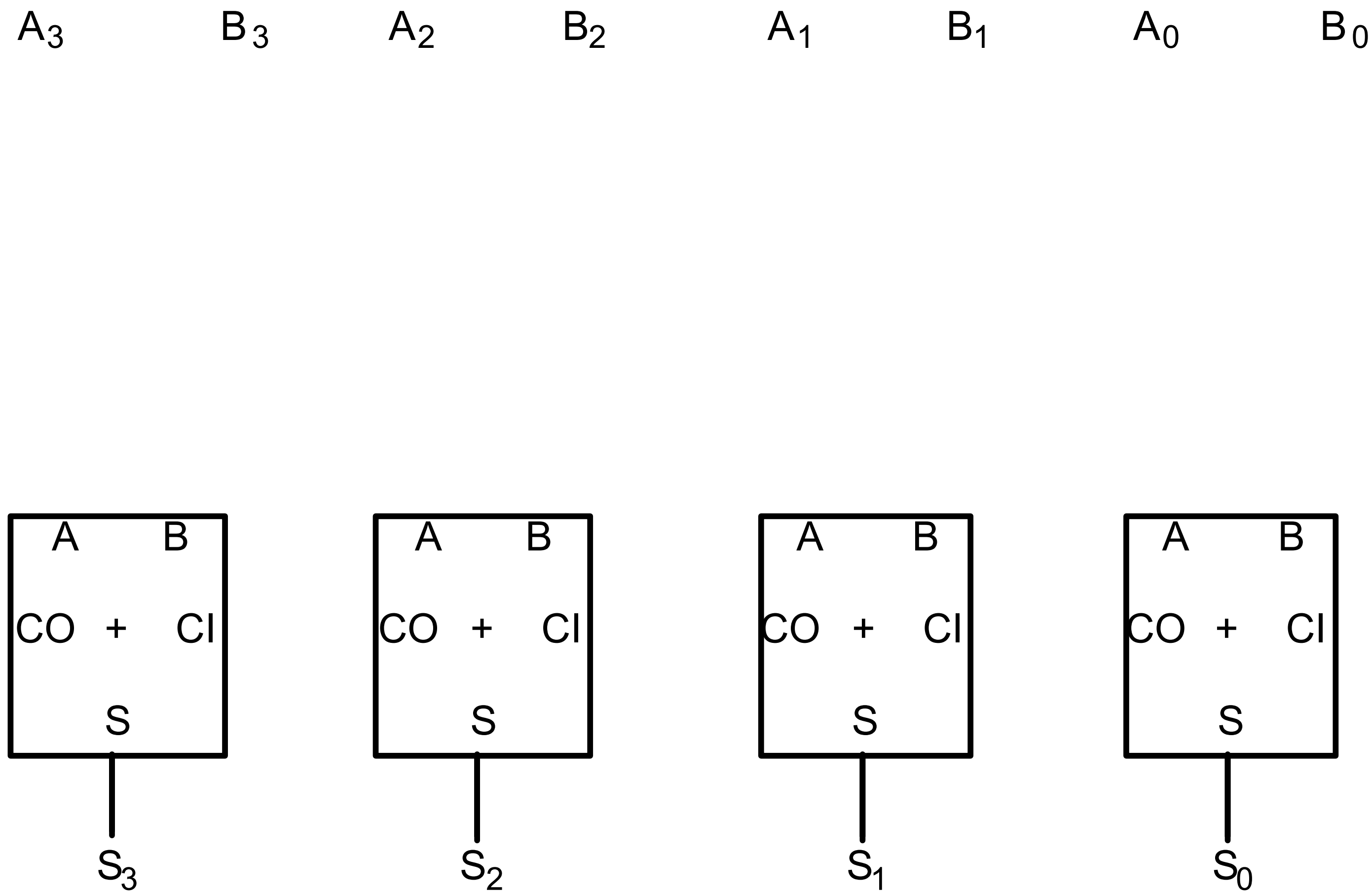
5	**0 1 0 1**	**-7**	**1 0 0 1**
3	**0 0 1 1**	**-2**	**1 1 1 0**
-8		**7**	
Overflow		**Overflow**	
5	**0 1 0 1**	**-3**	**1 1 0 1**
2	**0 0 1 0**	**-5**	**1 0 1 1**
7		**-8**	
No overflow		**No overflow**	

Full Adder

A	B	Cin	A+B+Cin	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

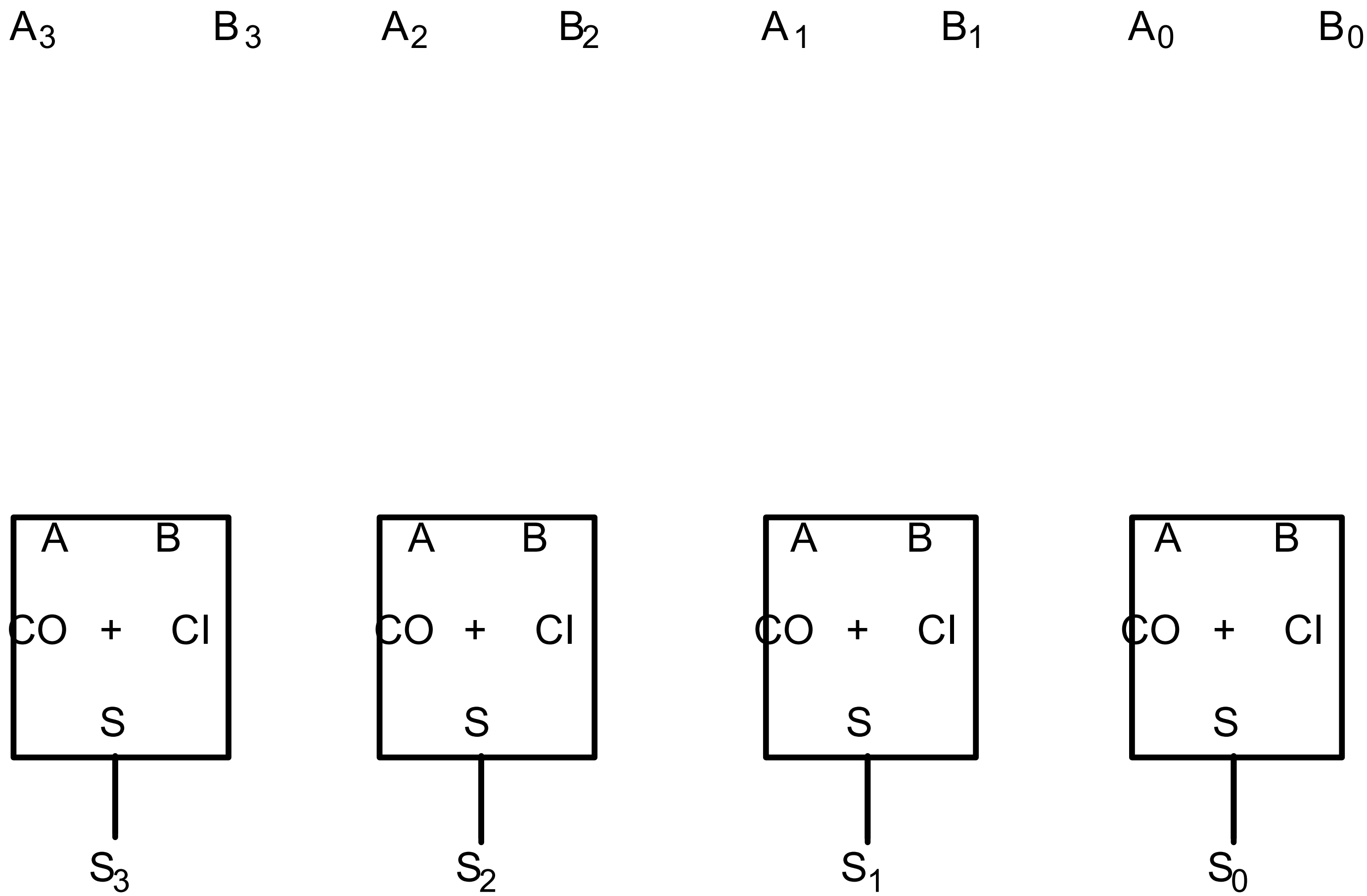
Multi-Bit Addition

$$\begin{array}{r} A_3\,A_2\,A_1\,A_0 \\ +\;\; B_3\,B_2\,B_1\,B_0 \\ \hline \end{array}$$



Adder/Subtractor

A + B =
A – B =

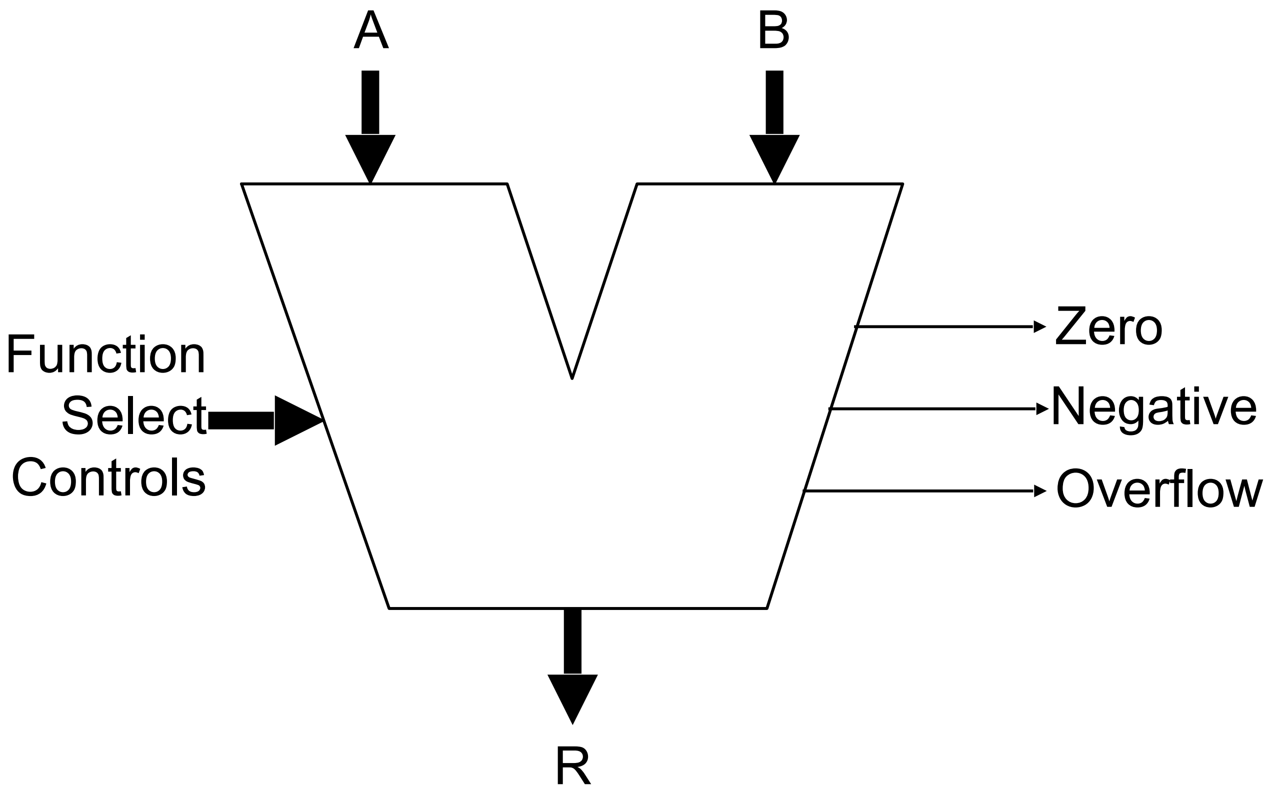


Overflow

ALU: Arithmetic Logic Unit

Computes arithmetic & logic functions based on controls

- Add, subtract
- XOR, AND, NAND, OR, NOR
- ==, <, overflow, …



Multiplication

Example

Multiplicand:	**0**	**1**	**1**	**0**	**6**
Multiplier:	**0**	**1**	**0**	**1**	**5**

4 partial products

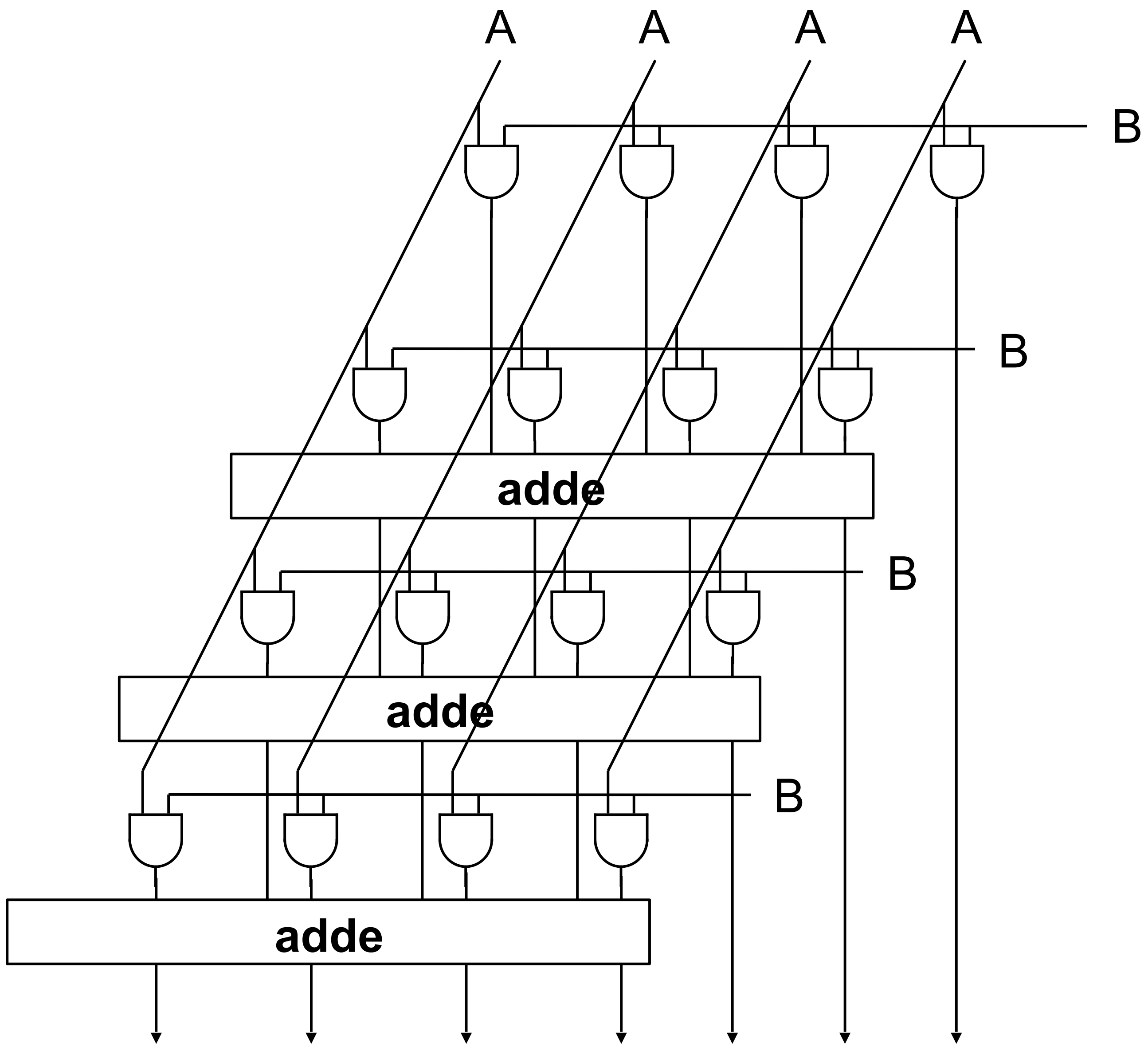
30

Repeat n times:

Compute partial product; shift; add

NOTE: Each bit of partial products is just an AND operation

Parallel Multipliers

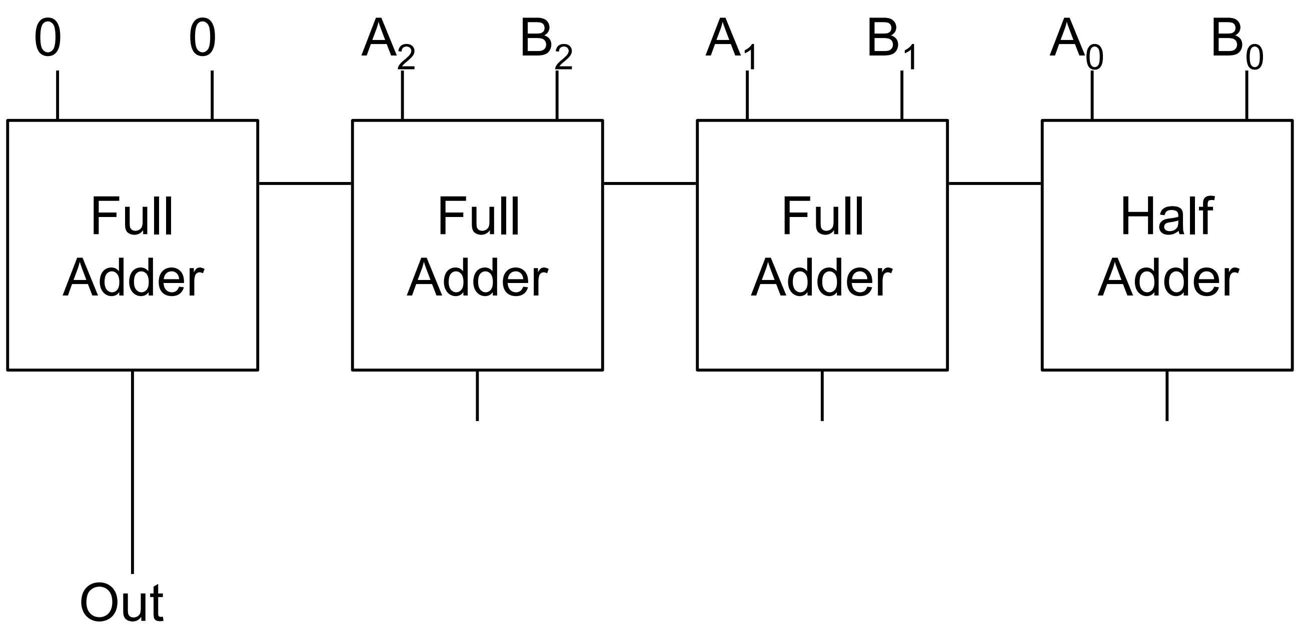


Debugging Complex Circuits

Complex circuits require careful debugging

Rip up and retry?

Ex. Circuit to see if A+B>7.



Debugging Complex Circuits (cont.)

```
module fullAdd (Cout, S, A, B, Cin);
  output Cout, S;  input A, B, Cin;

  assign Cout = (A&B) | (A&Cin) | (B&Cin);
  assign S = A^B^Cin;
endmodule

module halfAdd (Cout, S, A, B);
  output Cout, S;  input A, B;

  fullAdd a1(.Cout, .S, .A, .B, .Cin);
endmodule

module greaterThan7 (Out, A, B);
  output Out;  input [2:0] A, B;  wire [3:0] C, S;

  halfAdd pos0(.Cout(C[0]), .S(S[0]), .A(A[0]), .B(B[0]));
  fullAdd pos1(.Cout(C[1]), .S(S[1]), .A(A[1]), .B(B[1]), .C(C[0]));
  fullAdd pos2(.Cout(C[2]), .S(S[2]), .A(A[2]), .B(B[2]), .C(C[1]));
  fullAdd pos3(.Cout(C[3]), .S(Out), .A(0), .B(0), .C(C[2]));
endmodule
```

Debugging Approach

Test all behaviors.

    All combinations of inputs for small circuits, subcircuits.

Identify any incorrect behaviors.

Examine inputs and outputs to find earliest place where value is wrong.

    Typically, trace backwards from bad outputs, forward from inputs.

    Look at values at intermediate points in circuit.

DO NOT RIP UP, DEBUG!