Binary Number Systems and Arithmetic Circuits
Number systems

- Representation of positive numbers is the same in most systems
- Major differences are in how negative numbers are represented
- Representation of negative numbers come in three major schemes
  - sign and magnitude
  - 1s complement
  - 2s complement
  - excess code
- Assumptions
  - we'll assume a 4 bit machine word
  - 16 different values can be represented
  - roughly half are positive, half are negative
Sign and magnitude

- One bit dedicated to sign (positive or negative)
  - sign: 0 = positive (or zero), 1 = negative
- Rest represent the absolute value or magnitude
  - three low order bits: 0 (000) thru 7 (111)
- Range for n bits
  - +/- \((2^{n-1} - 1)\) (two representations for 0)
- Cumbersome addition/subtraction
  - must compare magnitudes to determine sign of result

0 100 = + 4
1 100 = - 4

Diagram showing binary values for sign and magnitude representation.
1s complement

- If \( N \) is a positive number, then the negative of \( N \) (its 1s complement or \( N' \)) is
  \[ N' = (2^n - 1) - N \]
- example: 1s complement of 7

\[
\begin{align*}
2^4 &= 10000 \\
-1 &= 00001 \\
2^4 - 1 &= 1111 \\
-7 &= 0111 \\
1000 &= -7 \text{ in 1s complement form}
\end{align*}
\]

- shortcut: simply compute bit-wise complement ( 0111 -> 1000 )
1s complement (cont'd)

- Subtraction implemented by 1s complement and then addition
- Two representations of 0
  - causes some complexities in addition
- High-order bit can act as sign bit

0 100 = + 4
1 011 = – 4
2s complement

- 1s complement with negative numbers shifted one position clockwise
  - only one representation for 0
  - one more negative number than positive numbers
  - high-order bit can act as sign bit

0 100 = + 4
1 100 = - 4
2s complement (cont’d)

- If N is a positive number, then the negative of N (its 2s complement or N*) is \( N^* = 2^n - N \)
  - example: 2s complement of 7
    \[
    \begin{align*}
    2^4 & = 10000 \\
    \text{subtract} & \quad 7 = 0111 \\
    & \quad 1001 = \text{repr. of } -7
    \end{align*}
    \]
  - example: 2s complement of \(-7\)
    \[
    \begin{align*}
    2^4 & = 10000 \\
    \text{subtract} & \quad -7 = 1001 \\
    & \quad 0111 = \text{repr. of } 7
    \end{align*}
    \]
  
- shortcut: 2s complement = bit-wise complement + 1
  - 0111 \rightarrow 1000 + 1 \rightarrow 1001 (representation of -7)
  - 1001 \rightarrow 0110 + 1 \rightarrow 0111 (representation of 7)
Addition and subtraction

- **Sign and Magnitude**

  When operands have the same sign, the result has the same sign as the operands:
  
<table>
<thead>
<tr>
<th>4</th>
<th>0100</th>
<th>-4</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>+3</td>
<td>0011</td>
<td>+(-3)</td>
<td>1011</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>-7</td>
<td>1111</td>
</tr>
</tbody>
</table>

  When signs differ, operation is subtract. Sign of result depends on the sign of the number with larger magnitude:
  
<table>
<thead>
<tr>
<th>4</th>
<th>0100</th>
<th>-4</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>1011</td>
<td>+ 3</td>
<td>0011</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>-1</td>
<td>1001</td>
</tr>
</tbody>
</table>
Addition and subtraction (cont'd)

- Ones' Complement

\[
\begin{array}{c|c|c|c}
& +0 & +1 & +2 \\
\hline
-0 & 0000 & 0001 & 0010 \\
-1 & 0011 & 0100 & 0101 \\
-2 & 0110 & 1000 & 1001 \\
-3 & 1110 & 1100 & 1101 \\
-4 & 1111 & 1010 & 1011 \\
-5 & 1010 & 0110 & 0111 \\
-6 & 1000 & 0010 & 0011 \\
-7 & 0000 & 0100 & 0101 \\
\end{array}
\]
Why does end-around carry work?
- It's equivalent to subtracting $2^n$ and adding 1

\[
M - N = M + N' = M + (2^n - 1 - N) = (M - N) + 2^n - 1 \quad (M > N)
\]

after end around carry:
\[= M - N\]

\[-M + (-N) = M' + N' = (2^n - M - 1) + (2^n - N - 1) \quad (M + N < 2^{n-1})\]

\[= 2^n + [2^n - 1 - (M + N)] - 1\]

after end around carry:
\[= 2^n - 1 - (M + N)\]

this is the correct form for representing $-(M + N)$ in 1s’ comp!
Addition and subtraction (cont'd)

- **2s complement**
  - simple scheme makes 2s complement the virtually unanimous choice for integer number systems in computers

![Circle diagram with binary numbers and operations]

<table>
<thead>
<tr>
<th>+3</th>
<th>0011</th>
<th>+ (-3)</th>
<th>1101</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0111</td>
<td>- 7</td>
<td>11001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>-3</th>
<th>1101</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 4</td>
<td>1100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>+3</th>
<th>0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10001</td>
</tr>
</tbody>
</table>

-1 0000 +0 0111 +1 0011 +2 0010 +3 0010 +4 0001 +5 0110 +6 0101 +7 0100 +8 0010 +9 0101 +10 0110 +11 0001

-2 1111 0000 +1 0011 +2 0010 +3 0010 +4 0001 +5 0110 +6 0101 +7 0100 +8 0010 +9 0101 +10 0110 +11 0001

11001101 11001 110000111111 1010 1001 1000 1010 1011 1101 1110 1111 0110 0101 0100 0010 0001
Why can the carry-out be ignored?

- Can't ignore it completely
  - needed to check for overflow (see next two slides)
- When there is no overflow, carry-out may be true but can be ignored

\[-M + N \text{ when } N > M:\]

\[M^* + N = (2^n - M) + N = 2^n + (N - M)\]

ignoring carry-out is just like subtracting \(2^n\)

\[-M + -N \text{ where } N + M \leq 2^{n-1}\]

\[(-M) + (-N) = M^* + N^* = (2^n - M) + (2^n - N) = 2^n - (M + N) + 2^n\]

ignoring the carry, it is just the 2s complement representation for \(- (M + N)\)
Overflow in 2s complement addition/subtraction

- **Overflow conditions**
  - add two positive numbers to get a negative number
  - add two negative numbers to get a positive number

5 + 3 = −8
−7 − 2 = +7
### Overflow conditions

- **Overflow when carry into sign bit position is not equal to carry-out**

\[
\text{ovf} = c_{n-1} \oplus c_n
\]

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

---

**Example 1:**

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

**Example 2:**

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Overflow

Overflow
Excess code

Used for floating point representation

<table>
<thead>
<tr>
<th>sign</th>
<th>exponent</th>
<th>significand</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>E</td>
<td>M</td>
</tr>
</tbody>
</table>

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

Excess 8 code
Arithmetic circuits

- Excellent examples of combinational logic design
- Time vs. space trade-offs
  - doing things fast may require more logic and thus more space
  - example: carry lookahead logic
- Arithmetic and logic units
  - general-purpose building blocks
  - critical components of processor datapaths
  - used within most computer instructions
Circuits for binary addition

- **Half adder** (add 2 1-bit numbers)
  - Sum = $A_i' B_i + A_i B_i' = A_i \oplus B_i$
  - Cout = $A_i B_i$

- **Full adder** (carry-in to cascade for multi-bit adders)
  - Sum = $C_i \oplus A \oplus B$
  - Cout = $B C_i + A C_i + A B = C_i (A + B) + A B = C_i (A \oplus B) + A B$

<table>
<thead>
<tr>
<th>$A_i$</th>
<th>$B_i$</th>
<th>Cin</th>
<th>Sum</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Full adder implementations

- Standard approach
  - 6 gates
  - 2 XORs, 2 ANDs, 2 ORs
Full adder implementations

- Alternative implementation
  - $\text{Cout} = A \times B + B \times \text{Cin} + A \times \text{Cin} = A \times B + \text{Cin} \times (A \oplus B)$
  - 5 gates
  - 2 XORs, 2 ANDs, 1 OR
  - two half adders and one OR gate
Adder/subtractor

- Use an adder to do subtraction thanks to 2s complement representation
  - $A - B = A + (-B) = A + B' + 1$
  - control signal selects B or 2s complement of B
Ripple-carry adders

- Critical delay
  - the propagation of carry from low to high order stages
  
  late arriving signal
  
  two gate delays to compute Cout

4 stage adder
1111+0000 --> 1111+0001

What if C0=1?

C0=0

A0 → S0 @2
B0 → C1 @2

A1 → S1 @3
B1 → C2 @4

A2 → S2 @5
B2 → C3 @6

A3 → S3 @7
B3 → Cout @8
Ripple-carry adders (cont’d)

- Critical delay
  - the propagation of carry from low to high order stages
  - $1111 + 0001$ is the worst case addition
  - carry must propagate through all bits
Carry-lookahead logic

- **Carry generate:** \( G_i = A_i B_i \)
  - must generate carry when \( A = B = 1 \)

- **Carry propagate:** \( P_i = A_i \oplus B_i \)
  - carry-in will equal carry-out here

- **Sum and Cout can be re-expressed in terms of generate/propagate:**
  - \( S_i = A_i \oplus B_i \oplus C_i \)
    \[= P_i \oplus C_i\]
  - \( C_{i+1} = A_i B_i + A_i C_i + B_i C_i \)
    \[= A_i B_i + C_i (A_i + B_i)\]
    \[= A_i B_i + C_i (A_i \oplus B_i)\]
    \[= G_i + C_i P_i\]
Carry-lookahead logic (cont’d)

- Re-express the carry logic as follows:
  - \( C_1 = G_0 + P_0 C_0 \)
  - \( C_2 = G_1 + P_1 C_1 = G_1 + P_1 G_0 + P_1 P_0 C_0 \)
  - \( C_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \)
  - \( C_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0 \)

- Each of the carry equations can be implemented with two-level logic
  - all inputs are now directly derived from data inputs and not from intermediate carries
  - this allows computation of all sum outputs to proceed in parallel
**Carry-lookahead implementation**

- Adder with propagate and generate outputs

```
  Ai  Bi
  |    |
  v    v
  G1  Si @ 2 gate delays
  |    |
  v    v
  C2  Gi @ 1 gate delay

  C0  P0  G0
  "C1 @ 3"

  C0  P0  P1  G0  P1
  "C2 @ 3"

  C0  P0  P1  G0  G1
  "C3 @ 3"
```

Increasingly complex logic for carries

V - Combinational Logic Case Studies
Contemporary Logic Design
Carry-lookahead logic generates individual carries
- sums computed much more quickly in parallel
- however, cost of carry logic increases with more stages
Carry-lookahead adder with cascaded carry-lookahead logic

- Carry-lookahead adder
  - 4 four-bit adders with internal carry lookahead
  - second level carry lookahead unit extends lookahead to 16 bits
Carry-select adder

- Redundant hardware to make carry calculation go faster
  - compute two high-order sums in parallel while waiting for carry-in
  - one assuming carry-in is 0 and another assuming carry-in is 1
  - select correct result once carry-in is finally computed
### Arithmetic logic unit design specification

**M = 0, logical bitwise operations**

<table>
<thead>
<tr>
<th>S1 S0</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>$F_i = A_i$</td>
<td>input $A_i$ transferred to output</td>
</tr>
<tr>
<td>0 1</td>
<td>$F_i = \neg A_i$</td>
<td>complement of $A_i$ transferred to output</td>
</tr>
<tr>
<td>1 0</td>
<td>$F_i = A_i \text{ xor } B_i$</td>
<td>compute XOR of $A_i$, $B_i$</td>
</tr>
<tr>
<td>1 1</td>
<td>$F_i = A_i \text{ xnor } B_i$</td>
<td>compute XNOR of $A_i$, $B_i$</td>
</tr>
</tbody>
</table>

**M = 1, C0 = 0, arithmetic operations**

<table>
<thead>
<tr>
<th>S1 S0</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>$F = A$</td>
<td>input $A$ passed to output</td>
</tr>
<tr>
<td>0 1</td>
<td>$F = \neg A$</td>
<td>complement of $A$ passed to output</td>
</tr>
<tr>
<td>1 0</td>
<td>$F = A + B$</td>
<td>sum of $A$ and $B$</td>
</tr>
<tr>
<td>1 1</td>
<td>$F = (\neg A) + B$</td>
<td>sum of $B$ and complement of $A$</td>
</tr>
</tbody>
</table>

**M = 1, C0 = 1, arithmetic operations**

<table>
<thead>
<tr>
<th>S1 S0</th>
<th>Function</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>$F = A + 1$</td>
<td>increment $A$</td>
</tr>
<tr>
<td>0 1</td>
<td>$F = (\neg A) + 1$</td>
<td>twos complement of $A$</td>
</tr>
<tr>
<td>1 0</td>
<td>$F = A + B + 1$</td>
<td>increment sum of $A$ and $B$</td>
</tr>
<tr>
<td>1 1</td>
<td>$F = (\neg A) + B + 1$</td>
<td>$B$ minus $A$</td>
</tr>
</tbody>
</table>

**logical and arithmetic operations**

not all operations appear useful, but “fall out” of internal logic
**Sample ALU – truth table**

<table>
<thead>
<tr>
<th>M</th>
<th>S1</th>
<th>S0</th>
<th>Ci</th>
<th>Ai</th>
<th>Bi</th>
<th>Fi</th>
<th>Ci+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Sample ALU – multi-level discrete gate logic implementation

12 gates
Arithmetic logic unit design (cont’d)

- Sample ALU – clever multi-level implementation

first-level gates

use S0 to complement Ai

S0 = 0 causes gate X1 to pass Ai
S0 = 1 causes gate X1 to pass Ai’

use S1 to block Bi

S1 = 0 causes gate A1 to make Bi go forward as 0
(don't want Bi for operations with Ai)
S1 = 1 causes gate A1 to pass Bi

use M to block Ci

M = 0 causes gate A2 to make Ci go forward as 0
(don't want Ci for logical operations)
M = 1 causes gate A2 to pass Ci

other gates

for M=0 (logical operations, Ci is ignored)

\[ F_i = S_1 \text{Bi xor} (S_0 \text{xor} A_i) \]

\[ = S_1'S_0' (A_i) + S_1'S_0 (A_i') + S_1 S_0' (A_i' B_i + A_i B_i) + S_1 S_0 (A_i' B_i' + A_i B_i) \]

for M=1 (arithmetic operations)

\[ F_i = S_1 \text{Bi xor} (S_0 \text{xor} A_i) \text{xor} C_i \]

\[ = C_i+1 = C_i (S_0 \text{xor} A_i) + S_1 \text{Bi} ((S_0 \text{xor} A_i) \text{xor} C_i) \]

just a full adder with inputs S0 xor Ai, S1 Bi, and Ci
Combinational multiplier

- Basic concept

Partial products

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>1101 (13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplier</td>
<td>* 1011 (11)</td>
</tr>
<tr>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>10001111 (143)</td>
<td></td>
</tr>
</tbody>
</table>

Product of 2 4-bit numbers is an 8-bit number
Combinational multiplier (cont’d)

- Partial product accumulation

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A3</td>
<td>A2</td>
<td>A1</td>
<td>A0</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>B3</td>
<td>B2</td>
<td>B1</td>
<td>B0</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>A2</td>
<td>A2</td>
<td>A1</td>
<td>A0</td>
</tr>
<tr>
<td>B0</td>
<td>B0</td>
<td>B0</td>
<td>B0</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>A3</td>
<td>B1</td>
<td>A2</td>
<td>B1</td>
</tr>
<tr>
<td>B1</td>
<td>B1</td>
<td>B1</td>
<td>B1</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>A3</td>
<td>B2</td>
<td>A2</td>
<td>B2</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>A3</td>
<td>B3</td>
<td>A2</td>
<td>B3</td>
</tr>
<tr>
<td>B3</td>
<td>B3</td>
<td>B3</td>
<td>B3</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>S7</td>
<td>S6</td>
<td>S5</td>
<td>S4</td>
</tr>
<tr>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>S3</td>
<td>S2</td>
<td>S1</td>
<td>S0</td>
</tr>
</tbody>
</table>
```
Combinational multiplier (cont’d)

Note use of parallel carry-outs to form higher order sums

12 Adders, if full adders, this is 6 gates each = 72 gates

16 gates form the partial products

total = 88 gates!
Combinational multiplier (cont’d)

- Another representation of the circuit

Building block: full adder + and

4 x 4 array of building blocks