Combinational Circuits in Processors
Arithmetic Circuits

Digital Design and Computer Architecture
Mohammad Sadrosadati
Frank K. Gürkaynak

http://safari.ethz.ch/ddca
In This Lecture

- Why are arithmetic circuits so important

- Adders
  - Adding two binary numbers
  - Adding more than two binary numbers
  - Circuits Based on Adders

- Multipliers

- Functions that do not use adders

- Arithmetic Logic Units
Motivation: Arithmetic Circuits

- Core of every digital circuit
  - Everything else is side-dish, arithmetic circuits are the heart of the digital system

- Determines the performance of the system
  - Dictates clock rate, speed, area
  - If arithmetic circuits are optimized performance will improve

- Opportunities for improvement
  - Novel algorithms require novel combinations of arithmetic circuits, there is always room for improvement
Example: ARM Microcontroller

- Most popular embedded micro controller.

- Contains:
  - Multiplier
  - Accumulator
  - ALU/Adder
  - Shifter
  - Incrementer
Example: ARM Instructions

<table>
<thead>
<tr>
<th>MOV</th>
<th>RSB</th>
<th>CMP</th>
<th>SMLAW</th>
<th>B</th>
<th>LDRSB</th>
<th>LD,STRD</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVN</td>
<td>RSC</td>
<td>CMN</td>
<td>CLZ</td>
<td>BL</td>
<td>LD,STRH</td>
<td>PLD</td>
</tr>
<tr>
<td>MRS</td>
<td>MUL</td>
<td>QADD</td>
<td>TST</td>
<td>BX</td>
<td>LDRSH</td>
<td>SWP</td>
</tr>
<tr>
<td>MSR</td>
<td>MLA</td>
<td>QDADD</td>
<td>TEQ</td>
<td>BLX</td>
<td>LD,STM</td>
<td>SWI</td>
</tr>
<tr>
<td>ADD</td>
<td>UMULL</td>
<td>QSUB</td>
<td>AND</td>
<td>LD,STR</td>
<td>LD,STMIB</td>
<td>BKPT</td>
</tr>
<tr>
<td>ADC</td>
<td>UMLAL</td>
<td>SMUL</td>
<td>XOR</td>
<td>LD,STRT</td>
<td>LD,STMIA</td>
<td>CDP</td>
</tr>
<tr>
<td>SUB</td>
<td>SMULL</td>
<td>SMULA</td>
<td>OR</td>
<td>LD,STRB</td>
<td>LD,STMDB</td>
<td>MRC,MCR</td>
</tr>
<tr>
<td>SBC</td>
<td>SMLAL</td>
<td>SMULW</td>
<td>BIC</td>
<td>LD,STRBT</td>
<td>LD,STMDA</td>
<td>MRRC,MCRR</td>
</tr>
</tbody>
</table>

This table lists various ARM instructions used in ARM architecture.
# Arithmetic Based Instructions of ARM

<table>
<thead>
<tr>
<th>MOV</th>
<th>RSB</th>
<th>CMP</th>
<th>SMLAW</th>
<th>B</th>
<th>LDREIB</th>
<th>LD,STRD</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVN</td>
<td>RSC</td>
<td>CMN</td>
<td>CLZ</td>
<td>BL</td>
<td>LD,STRH</td>
<td>PLD</td>
</tr>
<tr>
<td>MRIS</td>
<td>MUL</td>
<td>QADD</td>
<td>TST</td>
<td>BX</td>
<td>LDREH</td>
<td>SWP</td>
</tr>
<tr>
<td>MSR</td>
<td>MLA</td>
<td>QDADD</td>
<td>TEQ</td>
<td>BLX</td>
<td>LD,STM</td>
<td>SWI</td>
</tr>
<tr>
<td>ADD</td>
<td>UMULL</td>
<td>QSUB</td>
<td>ANG</td>
<td>L,D,STR</td>
<td>LD,STMIB</td>
<td>BKPT</td>
</tr>
<tr>
<td>ADC</td>
<td>UMLAL</td>
<td>SMUL</td>
<td>XOR</td>
<td>L,D,STRT</td>
<td>LD,STMIA</td>
<td>CDP</td>
</tr>
<tr>
<td>SUB</td>
<td>SMULL</td>
<td>SMULA</td>
<td>OR</td>
<td>L,D,STRB</td>
<td>LD,SYMDB</td>
<td>MRC,MCR</td>
</tr>
<tr>
<td>SBC</td>
<td>SMLAL</td>
<td>SMULW</td>
<td>BIC</td>
<td>L,D,STRBT</td>
<td>LD,STMDA</td>
<td>MRR,MCR</td>
</tr>
</tbody>
</table>

- **MOV**: Move
- **MVN**: Move Negation
- **MRIS**: Move Register to Integer
- **MSR**: Move Register to Register
- **ADD**: Add
- **ADC**: Add with Carry
- **SUB**: Subtract
- **SBC**: Subtract with Carry
- **RSC**: Reverse Subtraction
- **RSB**: Reverse Subtract
- **RSC**: Reverse Subtract
- **RSC**: Reverse Subtract
- **MUL**: Multiply
- **MLA**: Multiply with Accumulator
- **UMULL**: Unsigned Multiply
- **UMLAL**: Unsigned Multiply with Accumulator
- **SMULL**: Signed Multiply
- **SMLAL**: Signed Multiply with Accumulator
- **CMP**: Compare
- **CMN**: Compare Negation
- **QADD**: Quad Add
- **QDADD**: Quad Add with Accumulator
- **CLZ**: Clear Lower Zero
- **TST**: Test
- **TEQ**: Test Equal
- **AND**: And
- **XOR**: Exclusive Or
- **OR**: Or
- **BIC**: Bit Inversion Carry
- **B**: Branch
- **BL**: Branch Link
- **BX**: Branch to X
- **LD**: Load
- **STR**: Store
- **LDREH**: Load Register from Exception
- **LD,STM**: Load Store Memory
- **LD,STMIA**: Load Store Memory with Interrupt Acknowledge
- **LD,STMIB**: Load Store Memory with Interrupt Block
- **LD,STRBT**: Load Store Memory with Branch Take
- **LD,STRD**: Load Store Memory with Delay
- **LD,STRH**: Load Store Memory with Hold
- **LD,STRM**: Load Store Memory with Mask
- **LD,STRMB**: Load Store Memory with Mask Branch
- **LD,STRMT**: Load Store Memory with Test Take
- **LD,STRMIB**: Load Store Memory with Interrupt Block Acknowledge
- **LD,STMIA**: Load Store Memory with Interrupt Acknowledge
- **LD,STMIB**: Load Store Memory with Interrupt Block
- **LD,STMDB**: Load Store Memory with Delay Branch
- **LD,STMDA**: Load Store Memory with Delay Acknowledge
- **LD,STMDT**: Load Store Memory with Delay Test
- **LD,STMDM**: Load Store Memory with Mask Delay
- **LD,STMDMB**: Load Store Memory with Mask Delay Branch
- **LD,STMDMT**: Load Store Memory with Mask Delay Test
Types of Arithmetic Circuits

In order of complexity:

- Shift / Rotate
- Compare
- Increment / Decrement
- Negation
- Addition / Subtraction
- Multiplication
- Division
- Square Root
- Exponentation
- Logarithmic / Trigonometric Functions
Relation Between Arithmetic Operators
Addition

Addition is the *most important* operation in computer arithmetic. Our topics will be:

- Adding 1-bit numbers: Counting bits
- Adding two numbers: Basics of addition
- Circuits based on adders: Subtractors, Comparators
- Adding multiple numbers: Chains of Adders

Later we will also talk about fast adder architectures
Half-Adder (2,2) Counter

■ The *Half Adder* (HA) is the simplest arithmetic block

■ It can add two 1-bit numbers, result is a 2-bit number

■ Can be realized easily

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C₀</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

![Half Adder Diagram](image)
Full-Adder (3,2) Counter

- The Full Adder (FA) is the *essential* arithmetic block.
- It can add three 1-bit numbers, result is a 2-bit number.
- There are many realizations both at gate and transistor level.
- Since it is used in building many arithmetic operations, the performance of one FA influences the overall performance greatly.
Adding Multiple 1-bit Numbers

**linear structure**

**tree structure**
Adding Multiple Digits

- Similar to decimal addition
- Starting from the right, each digit is added
- The carry from one digit is added to the digit to the left

![Image showing decimal and binary addition examples]
Adding Multiple Digits

- Similar to decimal addition
- Starting from the right, each digit is added
- The carry from one digit is added to the digit to the left

Decimal

\[
\begin{array}{c}
\text{+} \\
1 \ 9 \ 1 \ 8 \\
4 \ 3 \ 7 \\
\hline
1 \ 3 \ 5 \ 5
\end{array}
\]

Binary

\[
\begin{array}{c}
\text{+} \\
1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \\
0 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \\
\hline
1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1
\end{array}
\]
Ripple Carry Adder (RCA)
Curse of the Carry

The most significant outputs of the adder depends on the least significant inputs

$n+1$ stages
Adding Multiple Numbers

- Multiple fast adders not a good idea
  - If more than 2 numbers are to be added, multiple fast adders are not really efficient

- Use an array of ripple carry adders
  - Popular and efficient solution

- Use carry save adder trees
  - Instead of using carry propagate adders (the adders we have seen so far), *carry save adders* are used to reduce multiple inputs to two, and then a single carry propagate adder is used to sum up.
Array of Ripple Carry Adders

A3 B3  A2 B2  A1 B1  A0 B0

FA    FA    FA    FA    HA

C3  C2  C1  C0

FA    FA    FA    FA    HA

D3  D2  D1  D0

FA    FA    FA    FA    HA

S5  S4  S3  S2  S1  S0
Carry Save Principle

- Reduces three numbers to two with a single gate delay

\[ C + S = E + F + G \]
Carry Save Principle

\[ Z = D + E + F + G + H \]

- An array of carry save adders reduce the inputs to two
- A final (fast) carry propagate adder (CPA) merges the two numbers
- Performance mostly dictated by CPA
Multipliers

- **Largest common arithmetic block**
  - Requires a lot of calculation

- **Has three parts**
  - Partial Product Generation
  - Carry Save Tree to reduce partial products
  - Carry Propagate Adder to finalize the addition

- **Adder performance (once again) is important**

- **Many optimization alternatives**
Decimal Multiplication

\[
\begin{array}{cccc}
2 & 4 & 1 & 7 \\
\times & 1 & 4 & 0 & 3 \\
\hline
7 & 2 & 5 & 1 \\
0 & 0 & 0 & 0 \\
9 & 6 & 6 & 8 \\
\hline
2 & 4 & 1 & 7 \\
\hline
3 & 3 & 9 & 1 & 0 & 5 & 1
\end{array}
\]
Binary Multiplication

\[
\begin{array}{ccccccccccc}
0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
\hline
0 & 1 & 1 & 0 & 1 & 1 & 1 \\
\hline
0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

Partial Products

\[4895\]
For n-bit Multiplier m-bit Multiplicand

- **Generate Partial Products**
  - For each bit of the multiplier the partial product is either
    - when ‘0’: all zeroes
    - when ‘1’: the multiplicand
  
  achieved easily by AND gates

- **Reduce Partial Products**
  - This is the job of a carry save adder

- **Generate the Result (n + m bits)**
  - This is a large, fast Carry Propagate Adder
Parallel Multiplier
Parallel Multiplier
Operations Based on Adders

Several well-known arithmetic operation are based on adders:

- Negator
- Incremerter
- Subracter
- Adder Subracter
- Comparator
Negating Two’s Complement Numbers

- To negate a two’s complement number
  
  $$-A = \overline{A} + 1$$

- All bits are inverted

- One is added to the result

- Can be realized easily by an adder.

- B input is optimized away
Incrementer

- B input is zero
- Carry In ($C_{in}$) of the adder can be used as the Increment (Inc) input
- Decrementer similar in principle
Subtractor

- B input is inverted
- $C_{in}$ of the adder is used to complement B
Subtractor

- B input is inverted
- \( C_{in} \) of the adder is used to complement B
- It can be made programmable so that both additions and subtractions can be performed at the same time

\[ Z = A \oplus B \]

\( C_o \)
Comparator

Based on a Subtractor

(A = B) = EQ

(A != B) = EQ

(A > B) = GE EQ

(A >= B) = GE

(A < B) = GE

(A <= B) = GE + EQ
Functions Realized Without Adders

- Not all arithmetic functions are realized by using adders
  - Shift / Rotate Units

- Binary Logic functions are also used by processors
  - AND
  - OR
  - XOR
  - NOT

These are implemented very easily
Shifters

- Logical shifter: shifts value to left or right and fills empty spaces with 0’s
  - Ex: 11001 >> 2 = ??
  - Ex: 11001 << 2 = ??

- Arithmetic shifter: same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
  - Ex: 11001 >>> 2 = ??
  - Ex: 11001 <<< 2 = ??

- Rotator: rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: 11001 ROR 2 = ??
  - Ex: 11001 ROL 2 = ??
Shifters

- **Logical shifter:** shifts value to left or right and fills empty spaces with 0’s
  - Ex: $11001 \gg 2 = 00110$
  - Ex: $11001 \ll 2 = 00100$

- **Arithmetic shifter:** same as logical shifter, but on right shift, fills empty spaces with the old most significant bit (msb).
  - Ex: $11001 \gg\gg 2 = 11110$
  - Ex: $11001 \ll\ll 2 = 00100$

- **Rotator:** rotates bits in a circle, such that bits shifted off one end are shifted into the other end
  - Ex: $11001 \text{ ROR } 2 = 01110$
  - Ex: $11001 \text{ ROL } 2 = 00111$
Shifter Design

\[ A_{3:0} \rightarrow \underbrace{4}_{\text{shamt}_{1:0}} \rightarrow 4 \rightarrow Y_{3:0} \]
Shifters as Multipliers and Dividers

- A left shift by $N$ bits multiplies a number by $2^N$
  - Ex: $00001 << 2 = 00100$ ($1 \times 2^2 = 4$)
  - Ex: $11101 << 2 = 10100$ ($-3 \times 2^2 = -12$)

- The **arithmetic** right shift by $N$ divides a number by $2^N$
  - Ex: $01000 >>> 2 = 00010$ ($8 \div 2^2 = 2$)
  - Ex: $10000 >>> 2 = 11100$ ($-16 \div 2^2 = -4$)
Other Functions

- **We have covered 90% of the arithmetic functions commonly used in a CPU**

- **Division**
  - Dedicated architectures not very common
  - Mostly implemented by existing hardware (multipliers, subtractors, comparators) iteratively

- **Exponential, Logarithmic, Trigonometric Functions**
  - Dedicated hardware (less common)
  - Numerical approximations:
    \[ \exp(x) = 1 + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots \]
  - Look-up tables (more common)
Arithmetic Logic Unit

The reason why we study digital circuits:
the part of the CPU that does something (other than copying data)

- Defines the basic operations that the CPU can perform directly
  - Other functions can be realized using the existing ones iteratively. (i.e. multiplication can be realized by shifting and adding)

- Mostly, a collection of resources that work in parallel.
  - Depending on the operation one of the outputs is selected
Example: Arithmetic Logic Unit (ALU), pg243

<table>
<thead>
<tr>
<th>$F_{2:0}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>001</td>
<td>A</td>
</tr>
<tr>
<td>010</td>
<td>A + B</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>A &amp; $\sim$B</td>
</tr>
<tr>
<td>101</td>
<td>A</td>
</tr>
<tr>
<td>110</td>
<td>A - B</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>
Example: ALU Design

<table>
<thead>
<tr>
<th>F_{2:0}</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>A &amp; B</td>
</tr>
<tr>
<td>001</td>
<td>A \mid B</td>
</tr>
<tr>
<td>010</td>
<td>A + B</td>
</tr>
<tr>
<td>011</td>
<td>not used</td>
</tr>
<tr>
<td>100</td>
<td>A &amp; \neg B</td>
</tr>
<tr>
<td>101</td>
<td>A \mid \neg B</td>
</tr>
<tr>
<td>110</td>
<td>A - B</td>
</tr>
<tr>
<td>111</td>
<td>SLT</td>
</tr>
</tbody>
</table>
Set Less Than (SLT) Example

- Configure a 32-bit ALU for the set if less than (SLT) operation. Suppose $A = 25$ and $B = 32$.
  - $A$ is less than $B$, so we expect $Y$ to be the 32-bit representation of 1 (0x00000001).
Configure a 32-bit ALU for the set if less than (SLT) operation. Suppose $A = 25$ and $B = 32$.

- A is less than B, so we expect $Y$ to be the 32-bit representation of 1 ($0x00000001$).
- For SLT, $F_{2:0} = 111$.
- $F_2 = 1$ configures the adder unit as a subtracter. So $25 - 32 = -7$.
- The two’s complement representation of -7 has a 1 in the most significant bit, so $S_{31} = 1$.
- With $F_{1:0} = 11$, the final multiplexer selects $Y = S_{31}$ (zero extended) = $0x00000001$
What Did We Learn?

- How can we add, subtract, multiply binary numbers

- What other circuits depend on adders
  - Subtracter
  - Incrementer
  - Comparator
  - Important part of Multiplier

- Other functions (shifting)

- How is an Arithmetic Logic Unit constructed