The basics of logic design

Yipeng Huang

Rutgers University

April 20, 2023
Table of contents

Announcements

Cache-friendly code
  Loop interchange
  Cache blocking
  Multilevel cache hierarchies
  Cache oblivious algorithms
  Memory hierarchy implications for software-hardware abstraction

Transistors: The building block of computers

Combinational logic
  Basic gates
  More-than-2-input gates

Functional completeness
  The set of logic gates \{\text{NOT, AND, OR}\} is universal
  The NAND gate is universal
  The NOR gate is universal
Announcements

Class session plan

▶ 4/20, 4/24, 4/27: Diving deeper: Digital logic. (CS:APP Chapter 4.2) (Recommended reading: Patterson & Hennessy, Computer organization and design, appendix on "The Basics of Logic Design." Available online via Rutgers Libraries)

▶ 5/1: Survey of advanced topics in computer architecture.
Table of contents

Announcements

Cache-friendly code
  Loop interchange
  Cache blocking
  Multilevel cache hierarchies
  Cache oblivious algorithms
  Memory hierarchy implications for software-hardware abstraction

Transistors: The building block of computers

Combinational logic
  Basic gates
  More-than-2-input gates

Functional completeness
  The set of logic gates \{\text{NOT, AND, OR}\} is universal
  The NAND gate is universal
  The NOR gate is universal
Cache-friendly code

Algorithms can be written so that they work well with caches

▶ Maximize hit rate
▶ Minimize miss rate
▶ Minimize eviction counts

Do so by:

▶ Increasing spatial locality.
▶ Increasing temporal locality.

Advanced optimizing compilers can automatically make such optimizations

▶ GCC optimizations
▶ -floop-interchange
▶ -floop-block
Loop interchange

Refer to textbook slides on "Rearranging loops to improve spatial locality"

- Loop interchange increases spatial locality.
- In PA5, fourth part "cacheBlocking" you can explore the impact of this on matrix multiplication.
- In practice, programmers do not have to worry about this optimization.
- Optimized automatically in GCC by compiler flag -floop-interchange and -O3.
Cache blocking

Refer to textbook slides on "Using blocking to improve temporal locality"

- Cache blocking increases temporal locality.
- In PA5, fourth part "cacheBlocking" you can explore the impact of this on matrix multiplication.
- In practice, programmers do not have to worry about this optimization.
- Optimized automatically in GCC by compiler flag `-floop-block`. But it is not part of default optimizations such as `-O3` so you have to remember to set it.
Multilevel cache hierarchies

Small fast caches nested inside large slow caches

- L1 data and instruction cache: 32KB, 64 set, 8-way associative, 64B block, 4 cycle latency.
- L2 cache: 256KB, 512 set, 8-way associative, 64B block, 10 cycle latency.
- L3 cache: 8MB, 8192 set, 16-way associative, 64B block, 40-75 cycle latency.

Notice how latency cost increases as $E$-way associativity increases.

Figure: Intel Core i7 cache hierarchy. Image credit CS:APP

Figure: Intel 2020 Gulftown die shot. Image credit AnandTech
Cache oblivious algorithms

The challenge in writing code / compiling programs to take advantage of caches:

▶ Programmers do not easily have information about target machine.
▶ Compiling binaries for every envisioned target machine is costly.
Matrix transpose baseline algorithm: iteration

\[
A = \begin{bmatrix}
  a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\
  a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\
  a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\
  a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3}
\end{bmatrix}
\]

\[
B = A^\top = \begin{bmatrix}
  a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} \\
  a_{0,1} & a_{1,1} & a_{2,1} & a_{3,1} \\
  a_{0,2} & a_{1,2} & a_{2,2} & a_{3,2} \\
  a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3}
\end{bmatrix}
\]

```c
for ( size_t i=0; i<n; i++ ) {
    for ( size_t j=0; j<n; j++ ) {
        B[ j*n + i ] = A[ i*n + j ];
    }
}
```
Matrix transpose via recursion

\[
A = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}
\]

\[
B = A^\top = \begin{bmatrix} A_{0,0}^\top & A_{0,1}^\top \\ A_{1,0}^\top & A_{1,1}^\top \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} \\ a_{0,1} & a_{1,1} & a_{2,1} & a_{3,1} \\ a_{0,2} & a_{1,2} & a_{2,2} & a_{3,2} \\ a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} \end{bmatrix}
\]

Strategy:

- Divide and conquer large matrix to transpose into smaller transpositions.
- After some recursion, problem will fit well inside cache capacity.
- Once enough locality exists within subroutine, switch to plain iterative approach.

Advantages:

- No need to know about cache capacity and parameters beforehand.
- Works well with deep multilevel cache hierarchies: different amounts of locality for each cache level.
Memory hierarchy implications for software-hardware abstraction

It is not entirely true the architecture can hide details of microarchitecture
Even less true going forward. What to do?

Application level recommendations

- Use industrial strength, optimized libraries compiled for target machine.
- Lapack, Linpack, Matlab, Python SciPy, NumPy...
- [https://people.inf.ethz.ch/markusp/teaching/263-2300-ETH-spring11/slides/class08.pdf](https://people.inf.ethz.ch/markusp/teaching/263-2300-ETH-spring11/slides/class08.pdf)

Algorithm level recommendations

Deploy cache-oblivious algorithm implementations.

Compiler level recommendations

- Enable compiler optimizations—*e.g.*, `-O3`, `-floop-interchange`, `-floop-block`.
Table of contents

Announcements

Cache-friendly code
  Loop interchange
  Cache blocking
  Multilevel cache hierarchies
  Cache oblivious algorithms
  Memory hierarchy implications for software-hardware abstraction

Transistors: The building block of computers

Combinational logic
  Basic gates
  More-than-2-input gates

Functional completeness
  The set of logic gates \( \{ \text{NOT, AND, OR} \} \) is universal
  The NAND gate is universal
  The NOR gate is universal
Computer organization
Layer cake

- Society
- Human beings
- Applications
- Algorithms
- High-level programming languages
- Interpreters
- Low-level programming languages
- Compilers
- Architectures
- Microarchitectures
- Sequential/combinational logic
- Transistors
- Semiconductors
- Materials science
Why binary

Everything is bits

- Each bit is 0 or 1
- By encoding/interpreting sets of bits in various ways
  - Computers determine what to do (instructions)
  - ... and represent and manipulate numbers, sets, strings, etc...

- Why bits? Electronic Implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires

To build logic, we need switches

Vacuum tubes a.k.a. valves

Figure: Source: By Stefan Riepl (Quark48) - Self-photographed, CC BY-SA 2.0

Transistors

The first transistor. Developed at Bell Labs, Murray Hill, New Jersey
https://www.bell-labs.com/about/locations/
MOSFETs

MOS: Metal-oxide-semiconductor
  ▶ A sandwich of conductor-insulator-semiconductor.

FET: Field-effect transistor
  ▶ Gate exerts electric field that changes conductivity of semiconductor.
**NMOS, PMOS, CMOS**

**PMOS: P-type MOS**
- Positive gate voltage, acts as open circuit (insulator)
- Negative gate voltage, acts as short circuit (conductor)

**NMOS: N-type MOS**
- Positive gate voltage, acts as short circuit (conductor)
- Negative gate voltage, acts as open circuit (insulator)

**CMOS: Complementary MOS**
- A combination of NMOS and PMOS to build logical gates such as NOT, AND, OR.
- We’ll go to slides posted in supplementary material to see how they work.
Combinational vs. sequential logic

Combinational logic
▶ No internal state nor memory
▶ Output depends entirely on input
▶ Examples: NOT, AND, NAND, OR, NOR, XOR, XNOR gates, decoders, multiplexers.

Sequential logic
▶ Has internal state (memory)
▶ Output depends on the inputs and also internal state
▶ Examples: latches, flip-flops, Mealy and Moore machines, registers, pipelines, SRAMs.
Table of contents

Announcements

Cache-friendly code
   Loop interchange
   Cache blocking
   Multilevel cache hierarchies
   Cache oblivious algorithms
   Memory hierarchy implications for software-hardware abstraction

Transistors: The building block of computers

Combinational logic
   Basic gates
   More-than-2-input gates

Functional completeness
   The set of logic gates \{\text{NOT, AND, OR}\} is universal
   The NAND gate is universal
   The NOR gate is universal
NOT gate

\[
\begin{array}{c|c}
A & \bar{A} \\
\hline
0 & 1 \\
1 & 0 \\
\end{array}
\]

**Table:** Truth table for NOT gate
**AND gate, NAND gate**

Table: Truth table for AND gate

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$AB$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table: Truth table for NAND gate

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$\overline{AB}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
OR gate, NOR gate

Table: Truth table for OR gate

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

Table: Truth table for NOR gate

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>( \overline{A + B} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
XOR gate, XNOR gate

Table: Truth table for XOR gate

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

Table: Truth table for XNOR gate

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
More-than-2-input AND gate

Table: Truth table for three-input AND gate

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$C$</th>
<th>$ABC$</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table: Truth table for three-input AND gate
More-than-2-input OR gate

Table: Truth table for three-input OR gate

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>$A + B + C$</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Table of contents

Announcements

Cache-friendly code
  Loop interchange
  Cache blocking
  Multilevel cache hierarchies
  Cache oblivious algorithms
  Memory hierarchy implications for software-hardware abstraction

Transistors: The building block of computers

Combinational logic
  Basic gates
  More-than-2-input gates

Functional completeness
  The set of logic gates \{\text{NOT, AND, OR}\} is universal
  The NAND gate is universal
  The NOR gate is universal
The set of logic gates \{\text{NOT, AND, OR}\} is universal.

\[
\begin{align*}
\text{And} & : & \text{Out} = a \land b \\
\text{Or} & : & \text{Out} = a \lor b \\
\text{Not} & : & \text{Out} = \neg a
\end{align*}
\]

\text{Figure: Source: CS:APP}
The set of logic gates \{\text{NOT}, \text{AND}, \text{OR}\} is universal

- Any truth table can be expressed as sum of products form.
- Write each row with output 1 as a product (minterm).
- Sum the products (minterm).
- Forms a disjunctive normal form (DNF).
- \( D = \overline{A}B\overline{C} + \overline{A}\overline{B}C \)
- Always only needs NOT, AND, OR gates.
- Supplementary slides example...

### Logical Completeness

Can implement ANY truth table with AND, OR, NOT.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

1. AND combinations that yield a "1" in the truth table.
2. OR the results of the AND gates.

Sum of products OR of AND clauses
The set of logic gates \{\text{NOT, AND, OR}\} is universal

- Any truth table can be expressed as sum of products form.
- Write each row with output 1 as a product (minterm).
- Sum the products (minterm).
- Forms a disjunctive normal form (DNF).
- \(D = \overline{A}BC + A\overline{B}C\)
- Always only needs NOT, AND, OR gates.
- Supplementary slides example...

**Figure: Source: CS:APP**
The NAND gate is universal

NOT gate as a single NAND gate

\[ A \rightarrow \overline{A} = \overline{A} \]

<table>
<thead>
<tr>
<th>( A )</th>
<th>( \overline{A} )</th>
<th>( AA )</th>
<th>( \overline{AA} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table: \( \overline{A} = \overline{AA} \)

AND gate as two NAND gates

\[ A \rightarrow \overline{AB} \rightarrow AB = \]

<table>
<thead>
<tr>
<th>( A )</th>
<th>( B )</th>
<th>( AB )</th>
<th>( \overline{AB} )</th>
<th>( \overline{AB} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table: \( AB = \overline{AB} \)
The NAND gate is universal

De Morgan’s Law

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$\overline{A}$</th>
<th>$\overline{B}$</th>
<th>$\overline{A} \overline{B}$</th>
<th>$A + B$</th>
<th>$\overline{A + B}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table: $\overline{A} \overline{B} = \overline{A + B}$

OR gate as three NAND gates

$$\begin{align*}
A \\
\overline{B}
\end{align*} = A + B$$
The NOR gate is universal

NOT gate as a single NOR gate

\[ A \rightarrow \overline{A} = A \rightarrow \overline{A} \]

<table>
<thead>
<tr>
<th>A</th>
<th>\overline{A}</th>
<th>A + A</th>
<th>\overline{A + A}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table: \( \overline{A} = A + A \)

OR gate as two NOR gates

\[ AB \rightarrow \overline{A + B} \rightarrow A + B \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A + B</th>
<th>A + B</th>
<th>\overline{A + B}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</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>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table: \( A + B = \overline{A + B} \)
The NOR gate is universal

De Morgan’s Law

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>$\overline{A}$</th>
<th>$\overline{B}$</th>
<th>$\overline{A} + \overline{B}$</th>
<th>AB</th>
<th>$\overline{AB}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<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>
<td>1</td>
<td>0</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>
</tr>
</tbody>
</table>

Table: $\overline{A} + \overline{B} = \overline{AB}$

AND gate as three NOR gates

$A \overline{B} = AB$

$\overline{A} + \overline{B} = AB$