ECE 5775 (Fall’17)
High-Level Digital Design Automation

Fixed-Point Types
Analysis of Algorithms
Announcements

▶ Lab 1 on CORDIC is released
  – Due Friday 9/8 @11:59am

▶ MEng TA: Jeffrey Witz (jmw483)
  – Office hour: Fridays 11:00am-12:00pm @ Rhodes 312
Q&A on Hardware Specialization, FPGA, HLS

▸ For CPU, does a memory access consume less energy than an ALU operation?

▸ Name three important forms of hardware specialization

▸ Which of the HLS optimizations is more expensive in area: Loop pipelining OR unrolling?

▸ To implement a 4:1 MUX with a single K-input LUT, what’s the minimum K? How many SRAM cells are required in the LUT?
A k-input LUT (k-LUT) can be configured to implement any k-input 1-output combinational logic
- $2^k$ SRAM bits
- Delay is independent of logic function
LUT Mapping Example

- Can we implement a 2:1 MUX using a network of 2-input LUTs?
Agenda

- Fixed-point types
  - Vivado HLS ap_int and ap_fixed classes

- Algorithm analysis
  - Complexity analysis and asymptotic notations
  - Taxonomy of algorithms
Additional Energy Savings from Specialization

- **Specialized memory architecture**
  - Exploit regular memory access patterns to minimize energy per memory read/write

- **Specialized communication architecture**
  - Exploit data movement patterns to optimize the structure/topology of on-chip interconnection network

- **Customized data type**
  - Exploit data range information to reduce bitwidth/precision and simply arithmetic operations

These techniques combined can lead to another 10-100X energy efficiency improvement over GPPs
Binary Number Representation

Unsigned number

- MSB has weight \(2^{n-1}\)
- Range of an \(n\)-bit unsigned number: ?

Two’s complement

- MSB has weight \(-2^{n-1}\)
- Range of an \(n\)-bit two’s complement number: ?

Examples: assuming integers here

<table>
<thead>
<tr>
<th>(2^3)</th>
<th>(2^2)</th>
<th>(2^1)</th>
<th>(2^0)</th>
<th>unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>= 11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(-2^3)</th>
<th>(2^2)</th>
<th>(2^1)</th>
<th>(2^0)</th>
<th>2’c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>= -5</td>
</tr>
</tbody>
</table>
Arbitrary Precision Integer in Vivado HLS

- C/C++ only provides a limited set of native integer types
  - char (8b), short (16b), int (32b), long (?), long long (64b)
  - Byte aligned: efficient in processors

- Arbitrary precision integer
  - Signed: `ap_int`; Unsigned `ap_uint`
  - Templatized class `ap_int<W>` or `ap_uint<W>`
    - W is the user-specified bitwidth
  - Two’s complement representation for signed integer

```c
#include "ap_int.h"
...
ap_int<9> x; // 9-bit
ap_uint<24> y; // 24-bit unsigned
ap_uint<512> z; // 512-bit unsigned
```
Representing Fractional Numbers

- Binary representation can also represent fractional numbers, usually called fixed-point numbers, by simply extending the pattern to include negative exponents
  - Less convenient to use compared to floating-point types
  - Efficient and cheap in application-specific hardware

\[
\begin{array}{ccccccc}
2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} & \text{unsigned} \\
1 & 0 & 1 & 1 & 0 & 1 & = 11.25 \\
\end{array}
\]

Binary point

\[
\begin{array}{ccccccc}
2'c \\
1 & 0 & 1 & 1 & 0 & 1 & = ? \\
\end{array}
\]
Overflow and Underflow

- **Overflow** occurs when a number is larger than the largest number that can be represented in a given number of bits.

<table>
<thead>
<tr>
<th>$2^5$</th>
<th>$2^4$</th>
<th>$2^3$</th>
<th>$2^2$</th>
<th>$2^1$</th>
<th>$2^0$</th>
<th>$2^{-1}$</th>
<th>$2^{-2}$</th>
<th>unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>= 11.25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>= 11.25</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td>= 11.25</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Drop MSB

- **Underflow** occurs when a number is smaller than the smallest number that can be represented.
Handling Overflow/Underflow

- One common (& efficient) way of handling overflow / underflow is to drop the most significant bits (MSBs) of the original number, often called \textit{wrapping}.

\[
\begin{array}{cccccc}
-2^3 & 2^2 & 2^1 & 2^0 & 2^{-1} & 2^{-2} \\
1 & 0 & 1 & 1 & 0 & 1 \\
\end{array} = -4.75
\]

Reduce integer width by 1
Wrap if overflows

\[
\begin{array}{cccccc}
0 & 1 & 1 & 0 & 1 \\
\end{array} = ?
\]

Wrapping can cause a negative number to become positive, or a positive to negative.
Fixed-Point Type in Vivado HLS

- Arbitrary precision fixed-point type
  - Signed: `ap_fixed`; Unsigned `ap_ufixed`
  - Templatized class `ap_fixed<W, I, Q, O>`
    - W: total word length
    - I: integer word length
    - Q: quantization mode
    - O: overflow mode
Example: Fixed-Point Modeling

- `ap_ufixed<11, 8, AP_TRN, AP_WRAP> x;`

- **MSB**
  - `b_7`...
  - `b_1` `b_0`...
  - `b_{-3}`

- **LSB**

- `binary point`

- 11 is the total number of bits in the type
- 8 bits to the left of the decimal point
- AP_TRN defines **truncation** behavior for quantization
- AP_WRAP defines **wrapping** behavior for overflow
Fixed-Point Type: Overflow Behavior

- **ap_fixed overflow mode**
  - Determines the behavior of the fixed point type when the result of an operation generates more precision in the **MSBs** than is available

  ```cpp
  ap_fixed<W, IW_X> x;
  ap_fixed<W, IW_Y> y = x; /* IW_Y < IW_X */
  ```

  **Default: AP_WRAP** (wrapping mode)

  **AP_SAT** (saturation mode)
Fixed-Point Type: Quantization Behavior

- **ap_fixed quantization mode**
  - Determines the behavior of the fixed point type when the result of an operation generates more precision in the **LSBs** than is available
  - Default mode: AP_TRN (truncation)
  - Other rounding modes: AP_RND, AP_RND_ZERO, AP_RND_INF, ...

```
ap_fixed<4, 2, AP_TRN> x = 1.25;   (b’01.01)
ap_fixed<3, 2, AP_TRN> y = x;
                   1.0   (b’01.0)
```

```
ap_fixed<4, 2, AP_TRN> x = -1.25;  (b’10.11)
ap_fixed<3, 2, AP_TRN> y = x;
                   -1.5  (b’10.1)
```
E-D-A Revisited

- **Exponential**
  - in complexity (or **Extreme** scale)

- **Diverse**
  - increasing system heterogeneity
  - multi-disciplinary

- **Algorithmic**
  - intrinsically computational
Analysis of Algorithms

- Need a systematic way to compare two algorithms
  - Runtime is often the most common criterion used
  - Space (memory) usage is also important in most cases
  - But difficult to compare in practice since algorithms may be implemented in different machines, use different languages, etc.
  - Additionally, runtime is usually input-dependent.

- big-O notation is widely used for asymptotic analysis
  - Complexity is represented with respect to some natural & abstract measure of the problem size \( n \)
Big-O Notation

▸ Express runtime as a function of input size $n$
  - Runtime $F(n)$ is of order $G(n)$, written as $F(n) = \mathcal{O}(G(n))$ when
    • $\exists n_0, \forall n \geq n_0, F(n) \leq KG(n)$ for some constant $K$
  - $F$ will not grow larger than $G$ by more than a constant factor
  - $G$ is often called an “upper bound” for $F$

▸ Interested in the worst-case input & the growth rate for large input size
How to determine the order of a function?
- Ignore lower order terms
- Ignore multiplicative constants

Examples:
\[ 3n^2 + 6n + 2.7 \text{ is } O(n^2) \]
\[ n^{1.1} + 10000000000n \text{ is } O(n^{1.1}), \ n^{1.1} \text{ is also } O(n^2) \]

- \( n! > C^n > n^c > \log n > \log\log n > C \)
  \[ \Rightarrow n > \log n, \ n \log n > n, \ n! > n^{10}. \]

What do asymptotic notations mean in practice?
- If algorithm A is \( O(n^2) \) and algorithm B is \( O(n \log n) \),
  \text{we usually say algorithm B is more scalable.}
Asymptotic Notions

- **big-Omega** notation $F(n) = \Omega(G(n))$
  - $\exists n_0, \forall n \geq n_0, F(n) \geq Kg(n)$ for some constant $K$
  - $G$ is called a “lower bound” for $F$

- **big-Theta** notation $F(n) = \Theta(G(n))$
  - if $G$ is both an upper and lower bound for $F$
  - Describes the growth of a function more accurately than $O(...)$ or $\Omega(...)$
  - Examples:
    - $4n^2 + 1024 = \Theta(n^2)$
    - $n^3 + 4n \neq \Theta(n^2)$
Exponential Growth

- Consider $2^n$, value doubled when $n$ is increased by 1

<table>
<thead>
<tr>
<th>n</th>
<th>$2^n$</th>
<th>1ns (/op) x $2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$10^3$</td>
<td>1 us</td>
</tr>
<tr>
<td>20</td>
<td>$10^6$</td>
<td>1 ms</td>
</tr>
<tr>
<td>30</td>
<td>$10^9$</td>
<td>1 s</td>
</tr>
<tr>
<td>40</td>
<td>$10^{12}$</td>
<td>16.7 mins</td>
</tr>
<tr>
<td>50</td>
<td>$10^{15}$</td>
<td>11.6 years</td>
</tr>
<tr>
<td>60</td>
<td>$10^{18}$</td>
<td>31.7 years</td>
</tr>
<tr>
<td>70</td>
<td>$10^{21}$</td>
<td>31710 years</td>
</tr>
</tbody>
</table>
NP-Complete

- The class **NP-complete** (NPC) is the set of decision problems which we “believe” there is no polynomial time algorithms (hardest problem in NP)

- **NP-hard** is another class of problems, which are at least as hard as the problems in NPC (also containing NPC)

- If we know a problem is in NPC or NP-hard, there is (very) little hope to solve it exactly in an efficient way
How to Identify an NP-Complete Problem

- I can’t find an efficient algorithm, I guess I’m just too dumb.
- I can’t find an efficient algorithm, because no such algorithm is possible.
- I can’t find an efficient algorithm, but neither can all these famous people.

[source: Computers and Intractibility by Garey and Johnson]
Reduction

- Showing a problem P is not easier than a problem Q
  - Formal steps:
    - Given an instance q of problem Q,
    - there is a polynomial-time transformation to an instance p of P
    - q is a “yes” instance iff p is a “yes” instance
  - Informally, if P can be solved efficiently, we can solve Q efficiently (Q is reduced to P)
    - P is polynomial time solvable $\rightarrow$ Q is polynomial time solvable
    - Q is not polynomial time solvable $\rightarrow$ P is not polynomial time solvable

- Example:
  - Problem A: Sort $n$ numbers
  - Problem B: Given $n$ numbers, find the median
Problem Intractability

- Most of the nontrivial EDA problems are intractable (NP-complete or NP-hard)
  - Best-known algorithm complexities that grow exponentially with \( n \), e.g., \( O(n!) \), \( O(n^n) \), and \( O(2^n) \).
  - No known algorithms can ensure, in a time-efficient manner, globally optimal solution

- **Heuristic** algorithms are used to find near-optimal solutions
  - Be content with a “reasonably good” solution
Many Algorithm Design Techniques

- There can be many different algorithms to solve the same problem
  - Exhaustive search
  - Divide and conquer
  - Greedy
  - Dynamic programming
  - Network flow
  - ILP
  - Simulated annealing
  - Evolutionary algorithms
  - ...
There are many ways to categorize different types of algorithms

- Polynomial vs. Exponential, in terms of computational effort
- Optimal (exact) vs. Heuristic, in terms of solution quality
- Deterministic vs. Stochastic, in terms of decision making
- Constructive vs. Iterative, in terms of structure
Broader List of Algorithms

- Combinatorial algorithms
  - Graph algorithms

- Computational mathematics
  - Optimization algorithms
  - Numerical algorithms

- Computational science
  - Bioinformatics
  - Linguistics
  - Statistics

- Information theory & signal processing

Graph Definition

- **Graph**: a set of objects and their connections
  - Importance: any binary relation can be represented as a graph

- **Formal definition**:
  - \( G = (V, E) \), \( V = \{v_1, v_2, ..., v_n\} \), \( E = \{e_1, e_2, ..., e_m\} \)
    - \( V \): set of **vertices** (nodes), \( E \): set of **edges** (arcs)
  - **Undirected graph**: if an edge \( \{u, v\} \) also implies \( \{v, u\} \)
  - **Directed graph**: each edge \( (u, v) \) has a direction
Simple Graph

- Loops, multi edges, and simple graphs
  - An edge of the form \((x, x)\) is said to be a **self-loop**
  - A graph permitted to have multiple edges (or parallel edges) between two vertices is called a **multigraph**
  - A graph is said to be **simple** if it contains no self-loops or multiedges

![Simple graph](image1)

![Multigraph](image2)
Graph Connectivity

- **Paths**
  - A **path** is any sequence of edges that connect two vertices
  - A **simple path** never goes through any vertex more than once

- **Connectivity**
  - A graph is **connected** if there is a path between any two vertices
  - Any subgraph that is connected can be referred to as a **connected component**
  - A directed graph is **strongly connected** if there is always a directed path between vertices
Next Class

- More graph algorithms
  - Timing analysis
  - BDDs