Computer Architecture for the Dark Silicon Era

Ali Mustafa Zaidi
PhD Student
University of Cambridge
Computer Laboratory
Brief History of Computer Architecture

[Diagram showing stages of instruction processing: Instruction Fetch, Decode/Register Fetch, Execute, Memory Access, Writeback]

[Graph showing IPC (Instructions Per Cycle) vs. core transistors with a trend line and a data point at 1M transistors]

Single Core
Brief History of Computer Architecture
Brief History of Computer Architecture

ILP Wall
- 7-12 limit
- ~ 9 currently

Memory Wall
- latency gap growing with frequency scaling

Complexity wall
- Due to exponential growth in resources

Power Wall (P = C.V^2.F)
- Cubic growth in power dissipation with frequency scaling
The Multicore Era

Power Wall

Memory Wall

Complexity Wall

ILP Wall
Multicore Programmability

- Programming more difficult because:
  - Concurrency statically exposed and managed by programmer (e.g. pthreads, MPI, OpenMP)
  - Nondeterministic parallel programming model very difficult to exploit
- Parallel programming mostly relegated to Embedded or HPC domains:
  - Where parallelism is easy to exploit, AND/OR
  - Advantages outweigh the costs.

Amdahl's Law & Speedup

\[ \text{Speedup}_{\text{Sym}}(f, q) = \frac{1}{\frac{(1-f)}{S_U(q)} + \frac{f}{N_{\text{Sym}}(q)S_U(q)}} \]

Fine Grained Multicore Scaling

Medium Grained Multicore Scaling

Coarse Grained Multicore Scaling
The Utilization Wall

<table>
<thead>
<tr>
<th>Param.</th>
<th>Description</th>
<th>Relation</th>
<th>Classical Scaling</th>
<th>Leakage Limited</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>power budget</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>chip size</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$V_t$</td>
<td>threshold voltage</td>
<td>$1/S$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$V_{dd}$</td>
<td>supply voltage</td>
<td>$\sim V_t \times 3$</td>
<td>$1/S$</td>
<td>1</td>
</tr>
<tr>
<td>$t_{ox}$</td>
<td>oxide thickness</td>
<td></td>
<td>$1/S$</td>
<td>1</td>
</tr>
<tr>
<td>W, L</td>
<td>transistor dimensions</td>
<td>$1/S$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$I_{sat}$</td>
<td>saturation current</td>
<td>$WV_{dd}/t_{ox}$</td>
<td>$1/S$</td>
<td>1</td>
</tr>
<tr>
<td>$p$</td>
<td>device power at full frequency</td>
<td>$I_{sat}V_{dd}$</td>
<td>$1/S^2$</td>
<td>1</td>
</tr>
<tr>
<td>$C_{gate}$</td>
<td>capacitance</td>
<td>$WL/t_{ox}$</td>
<td>$1/S$</td>
<td>$1/S$</td>
</tr>
<tr>
<td>$F$</td>
<td>device frequency</td>
<td>$I_{sat}/C_{gate}V_{dd}$</td>
<td>$S$</td>
<td>$S$</td>
</tr>
<tr>
<td>$D$</td>
<td>devices per chip</td>
<td>$A/(WL)$</td>
<td>$S^2$</td>
<td>$S^2$</td>
</tr>
<tr>
<td>$P$</td>
<td>full die, full frequency power</td>
<td>$D \times p$</td>
<td>1</td>
<td>$S^2$</td>
</tr>
<tr>
<td>$U$</td>
<td>utilization at fixed power</td>
<td>$B/P$</td>
<td>1</td>
<td>$1/S^2$</td>
</tr>
</tbody>
</table>

Amdahl's Law + Utilization Wall = Dark Silicon

Combination of Amdahl's Law and Utilization Wall severely limits multicore/manycore speed up from 45nm to 8nm process generations:

- CPU: 3.5x, GPU 2.4x (Conservative Scaling)
- CPU: 7.9x, GPU 2.7x (ITRS Scaling)
- Ideal speed up with Moore's law: 32x

Custom & Reconfigurable Computing

Energy Efficiency improvements over CPU:

- ASIC: 100-1000x
- FPGA: 10-100x
  - 100-1000x for domain-specific FPGAs
Energy Efficiency of Custom Computing

Why is custom computing energy efficient?

\[
EPI_{CPU} = \frac{E_{I-CacheWarmup}}{InstructionCount_{Static}} + \frac{(E_{Execute} + E_{CPUOverheads})}{InstructionCount_{Dynamic}}
\]

\[
E_{CPUOverheads} = E_{I-Cache} + E_{BranchPred} + E_{Decode} + E_{RegRename} + E_{Scheduling} + E_{Forwarding} + E_{RegFile} + E_{ROB}
\]
Why is custom computing energy efficient?

\[ E_{PI_{CPU}} = \frac{E_{I - \text{Cache Warmup}}}{\text{InstructionCount}_\text{Static}} + \frac{\left( E_{\text{Execute}} + E_{\text{CPU Overheads}} \right)}{\text{InstructionCount}_\text{Dynamic}} \]

\[ E_{CPU Overheads} = E_{I - \text{Cache}} + E_{\text{BranchPred}} + E_{\text{Decode}} + E_{\text{Reg Rename}} + E_{\text{Scheduling}} + E_{\text{Forwarding}} + E_{\text{Reg File}} + E_{\text{ROB}} \]

\[ E_{PI_{FPGA}} = \frac{E_{\text{Reconfiguration}}}{\text{InstructionCount}_\text{Static}} + \frac{\left( E_{\text{Execute}} + E_{\text{Multiplexing}} + E_{\text{Control}} + E_{\text{RC Overhead}} \right)}{\text{InstructionCount}_\text{Dynamic}} \]

\[ E_{I - \text{Cache Warmup}} \leq E_{\text{Reconfiguration}} \]

\[ E_{\text{CPU Overheads}} \gg E_{\text{Multiplexing}} + E_{\text{Control}} + E_{\text{RC Overhead}} \]
Energy Efficiency of Custom Computing

Why is custom computing energy efficient?

\[
EPI_{CPU} = \frac{E_{I-CacheWarmup}}{\text{InstructionCount}_{\text{Static}}} + \frac{(E_{\text{Execute}} + E_{\text{CPUOverheads}})}{\text{InstructionCount}_{\text{Dynamic}}}
\]

\[
E_{\text{CPUOverheads}} = E_{I-Cache} + E_{\text{BranchPred}} + E_{\text{Decode}} + E_{\text{RegRename}} + E_{\text{Scheduling}} + E_{\text{Forwarding}} + E_{\text{RegFile}} + E_{\text{ROB}}
\]

\[
EPI_{FPGA} = \frac{E_{\text{Reconfiguration}}}{\text{InstructionCount}_{\text{Static}}} + \frac{(E_{\text{Execute}} + E_{\text{Multiplexing}} + E_{\text{Control}} + E_{\text{RC-Overhead}})}{\text{InstructionCount}_{\text{Dynamic}}}
\]

\[
EPI_{ASIC} = \frac{(E_{\text{Execute}} + E_{\text{Multiplexing}} + E_{\text{Control}})}{\text{InstructionCount}_{\text{Dynamic}}}
\]

\[
E_{\text{I-CacheWarmup}} \leq E_{\text{Reconfiguration}}
\]

\[
E_{\text{CPUOverheads}} \gg E_{\text{Multiplexing}} + E_{\text{Control}} + E_{\text{RC-Overheads}}
\]
Issues with Custom Computing

• Productivity

  – Programmers require knowledge of hardware design (e.g. I/O timing, loop scheduling issues)

  – Use restricted subset of existing HLLs (no OOP, recursion, dynamic memory allocation)

  – Not suitable for rapid-recompilation based software engineering methodologies

  – Debugging & functional verification is often difficult.
Issues with Custom Computing

- Amenability
  - Poor performance on complex control-flow code
    - No Branch Prediction or Misprediction recovery mechanisms
    - If-conversion for forward branches...
    - Loop Unrolling, Flattening, Pipelining
      - Increases FSM complexity
  
- Poor adaptation to dynamically varying behaviour
  - Variable latency operations, e.g. Cache misses, Floating point ops, function calls.
  - Dynamically changing control-flow patterns.
Surviving in the Dark Silicon Era

• Scale performance
  – Increase energy efficiency (live with utilization wall)
  – Accelerate sequential code (break ILP wall)
  – Effectively utilize 'Dark' silicon.

• Reduce hardware costs
  – Reduce architecture complexity
  – Increase yields and (hard) fault tolerance
Thriving in the Dark Silicon Era!

- Scale performance
  - Increase energy efficiency (live with utilization wall)
  - Accelerate sequential code (break ILP wall)
  - Effectively utilize 'Dark' silicon.

- Reduce hardware costs
  - Reduce architecture complexity
  - Increase yields and (hard) fault tolerance

- Maintain (or Improve?) programmability
  - Support legacy applications and imperative programming model
    - bring back the 'free ride'?
  - Support existing and future concurrent programming models
Proposal

To achieve these goals, two key changes proposed for custom computing:

1) Switch from static to dynamic execution scheduling models, like the 'Spatial Computation' model proposed by Budiu et al.

2) Switch from Control-Data Flow Graph (CDFG) based Compiler IR to Value-State Dependence Graph (VSDG) based IR.
Dynamic Scheduling?

Dynamic Scheduling

Dynamic

Static

Execution Scheduling

Static

Dynamic

DPSE

SPSE

VLIW

Custom Computation

SPDE

Spatial Computation

Wavescalar

Superscalar

In Order

Superscalar

In Order

Custom Computation

Spatial Computation

EPIC

DPDE
Dynamic Scheduling

Increasing Compiler Complexity

Increasing Hardware Complexity

Dynamic

Static

Static

Dynamic

DPSE

EPIC

VLIW

Custom Computation

Superscalar

In Order

Wavescalar

Spatial Computation

Execution Scheduling
Why Dynamic Scheduling?

Increasing Compiler Complexity

Increasing Hardware Complexity

But also increasing performance on irregular code!
Custom vs Spatial Computation

Custom Computing

High Level Language

Graphical Intermediate Representation (e.g. CDFG)

Scheduling

Allocation & Binding

Synthesis

Centralized FSM

Statically Scheduled Custom Hardware (SPSE)
Custom vs Spatial Computation

Custom Computing
- High Level Language
- Graphical Intermediate Representation (e.g. CDFG)
- Scheduling
- Allocation & Binding
- Synthesis
- Centralized FSM (Statically Scheduled Custom Hardware (SPSE))

Spatial Computing
- High Level Language
- Graphical Intermediate Representation (e.g. CDFG)
- Allocation & Binding
- Synthesis
- Static Dataflow Machine in Hardware (SPDE)
**Introduction to Spatial Computation**

Program:

```plaintext
x = a & 7;
...
y = x >> 2;
```

Operations | Nodes | Pipeline stages
--- | --- | ---
Variables | Def-use edges | Channels (wires)

No interpretation

---

Advantages of Spatial Computation

- SPDE model more tolerant of variable delays
  - Easier to tolerate dynamically changing behaviour
  - Much lower programmer input required to manage I/O & Memory hierarchy

- Additionally, Budiu et al demonstrated:
  - Incorporation of conventional memory hierarchy
  - Full support for C language, including function calls, recursion, dynamic memory allocation
Advantages of Spatial Computation

- Compiler becomes much simpler
  - No scheduling, allocation, binding – Program dataflow directly implemented in hardware
  - Facilitates rapid recompilation, high-level debugging.

- Hardware becomes simpler
  - No centralized FSM
  - Can be implemented as asynchronous circuit – no clock needed
  - Simple, point-to-point interconnect
  - High-level performance/energy estimations more accurate?
Potential Disadvantages

- Increased area cost?
  - Dark Silicon: we have transistors to spare

- Low hardware utilization
  - No on-die hotspots?
  - Performance scaling & energy efficiency are more important
  - larger area (?) → more static power losses
    - Consider dynamic power-gating schemes.

- Energy Efficiency vs Custom Computing?
Remaining Issues

- Switching to the SPDE Spatial Computation model solves a lot of productivity problems, but
  - Currently supports only the C language
  - Support for advanced features such as OOP (C++), Streaming, Functional languages not demonstrated.

- SPDE alone does not resolve the Amenability problem
  - Complex control-flow code performance still poor
  - How do we enable aggressive speculation without introducing a centralized architectural bottleneck?
A Little more Perspective on ILP

David Wall (1991)
- Conventional Superscalar Processors
  - 7-12 ILP Max
    - With Best case BP, AA, RR

Lam, Wilson (1993)
- Control-flow primary limitation on ILP
  - “Multiple Flows of Control”
    - 5-20x ILP, even on 'sequential' code
Mak, Mycroft (2009)

- 10x ILP by excluding Control dependencies
  - e.g. via Speculation & Multiple Flows of Control
- Additional 10x ILP with perfect Memory Disambiguation!
A Little more Perspective on ILP

Multiple Flows of Control

```
for (i = 0; i < 100; i++)
  if (A[i] > 0) foo();
  bar();
```

Wavescalar Architecture
- Dynamic Dataflow, MFC, No BP or MLP
- 4-7x IPC over superscalar

Multiscalar Architecture
- 2-3x ILP over superscalar
Why the VSDG instead of the CDFG?

CDFG:
- Composed of multiple basic-blocks, each containing a DAG representing data flow within the block
- Control-dependences are explicit
- Data-flow is implicit
- Single Flow of Control!

VSDG:
- A single hierarchical DAG, representing data-dependences across the entire code
- Data-dependences are explicit
- Control-flow is implicit
- Multiple Flows of Control Achievable!
Why the VSDG instead of the CDFG?

CDFG: for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
    bar();

VSDG:

Start

i = 0

if (A[i] > 0) foo();
bar();

End
Why the VSDG instead of the CDFG?

CDFG

```plaintext
for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
    bar();
```

VSDG

```plaintext
i = 0
A
STATE_IN

= A[i]
> 0
foo()

i++
< 100
Next iteration of 'for' loop
FP T

STATE_OUT
```
Why the VSDG instead of the CDFG?

VSDG:

- Enables aggressive speculation via if-conversion of both forward branches and loops!
  - Loops represented as infinite DAGs
  - Physical implementation requires reintroduction of 'cycles', after appropriate degree of unrolling

- Enables Multiple Flows of Control
  - Loops and function-calls represented as 'complex' operations within the same dataflow graph

- Enables Independent pipelining of nested loops.
Verification

```c
/*
 * internal_int_transpose()
 * In-place (integer) matrix transpose algorithm.
 */
internal_int_transpose( mat, rows, cols )
    register int *mat;
    register int cols;
    int rows;
    {
        register int modulus = rows*cols - 1;
        register int swap_pos, current_pos, swap_val;

        /* loop, ignoring first and last elements */
        for (current_pos=1; current_pos<modulus; current_pos++)
            {
                /* Compute swap position */
                swap_pos = current_pos;
                do
                {
                    swap_pos = (swap_pos * cols) % modulus;
                } while (swap_pos < current_pos);

                if ( current_pos != swap_pos )
                    {
                        swap_val = mat[swap_pos];
                        mat[swap_pos] = mat[current_pos];
                        mat[current_pos] = swap_val;
                    }
            }
*/
```

- kernel from 'epic' benchmark (Mediabench)
- Nested loop with low ILP
  - Poor performance on Spatial Computation by Budiu et al.
- Hand-converted:
  - LLVM-IR → VSDG → 'VSFG' → Bluespec SystemVerilog
Performance

`internal_int_transpose()`

- Loops unrolled independently!
  - Inner loop: x1 (no unrolling)
  - Outer loop: x1, x2, x4
- Compared to:
  - 'Pegasus' SC
  - 4-way OOO Superscalar
  - 1-issue, in-order MIPS64

for 254 outer-loop iterations
Resource Requirements

- **internal_int_transpose()**
  - RTL-level estimation on Xilinx Spartan 6 Family FPGAs:
    - VSFG x1, x2, x4
    - Cheri MIPS64
  - Extrapolated results:
    - Optimised Cheri MIPS64

---

### Resource Utilization

<table>
<thead>
<tr>
<th></th>
<th>Reg</th>
<th>LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSFG x1</td>
<td>100</td>
<td>500</td>
</tr>
<tr>
<td>VSFG x2</td>
<td>200</td>
<td>1000</td>
</tr>
<tr>
<td>VSFG x4</td>
<td>500</td>
<td>1500</td>
</tr>
<tr>
<td>Cheri MIPS-64</td>
<td>1500</td>
<td>2000</td>
</tr>
<tr>
<td>Optimized MIPS-64</td>
<td>1000</td>
<td>1500</td>
</tr>
</tbody>
</table>
Power (preliminary results)

Resource Utilization estimated from Xilinx PlanAhead RTL Resource Estimation tool.

Activity Ratios assumed equal for all designs
- 12.5% for LUTs and Registers
- In reality: \( A_{Spatial\ Computation} \ll A_{MIPS64} \)

\[ \text{Power} = f(R, A) \]
- \( R \) = Resource Utilization
- \( A \) = Activity Ratios

Resource Utilization estimated from Xilinx PlanAhead RTL Resource Estimation tool.
Power (preliminary results)

Total Power Dissipation

- VSFG x1
- VSFG x2
- VSFG x4
- Cheri MIPS-64
- Optimized MIPS-64

Resource Utilization

- Logic
- BRAM
- Static
- Dynamic Clock

- LUT
- Reg
Energy (preliminary results)

Energy of whole computation

- \( Energy = P \times \left( \frac{1}{F} \right) \times Num_{cycles} \)
  - \( P = \) Power
  - \( F = \) Frequency (assumed 100MHz)

- Frequency assumed equal for all designs

- VSFG energy is ~ constant with performance increase!
  - Asynchronous implementation may improve Energy efficiency further!
Energy (preliminary results)

Energy of whole computation

Total Cycle Count

Total Power Dissipation

- Energy of whole computation
- Total Cycle Count
- Total Power Dissipation

Legend:
- Red: Energy
- Orange: Logic
- Yellow: BRAM
- Green: Static
- Blue: Dynamic Clock
Conclusion & Future Work

• Work Remaining:
  – Complete compiler toolchain.
  – Complete proper validation:
    • Post P&R comparison vs. Altera Nios II soft-processor (highly optimised architecture)
    • Compare with existing HLS tools for both FPGA and ASIC synthesis

• Future Work:
  – Incorporate MLP support (static & dynamic)
  – Develop Custom FPGA
    • Can we achieve ASIC level efficiencies on a programmable fabric?
  – Support concurrent programming models
  – Runtime scheduling of dataflow graphs on finite programmable resources.
Thank you
Additional Slides
Why the VSDG instead of the CDFG?

CDFG

```plaintext
for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
    bar();
```

VSDG
Why the VSDG instead of the CDFG?

CDFG

VSDG

for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
    bar();

Next iteration of 'for' loop

CPP

\[ \text{for (i = 0; i < 100; i++)} \]
\[ \text{if (A[i] > 0) foo();} \]
\[ \text{bar();} \]
Why the VSDG instead of the CDFG?

CDFG

for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
    bar();

VSDG

Next iteration of 'for' loop
Why the VSDG instead of the CDFG?

**CDFG**

for (i = 0; i < 100; i++)
    if (A[i] > 0) foo();
    bar();

**VSDG**

Next iteration of 'for' loop

End

Start

i = 0

STATE_IN

i

= A[i]

> 0

= A[i]

> 0

foo()
Pegasus IR

VSDG
Value State Dependence Graph

- Low overhead static loop 'pipelining':
  - The VSDG loop representation can be 'flattened' to increase parallelism.
Value State Dependence Graph

- Enabling outer loop pipelining:
  - Unlike Pegasus, No partitioning+sequencing
  - Instead, inner loop represented as nested subgraph.

```c
/*
================================================================================================
In-place (integer) matrix transpose algorithm.
================================================================================================*/

internal_int_transpose(mat, rows, cols)
  register int *mat;
  register int cols;
  int rows;
{
  register int modulus = rows*cols - 1;
  register int swap_pos, current_pos, swap_val;

  /* loop, ignoring first and last elements */
  for (current_pos=1; current_pos<modulus; current_pos++)
  {
    /* Compute swap position */
    swap_pos = current_pos;
    do
      
      { swap_pos = (swap_pos * cols) % modulus; }
    while (swap_pos < current_pos);

    if ( current_pos != swap_pos )
      { swap_val = mat[swap_pos];
        mat[swap_pos] = mat[current_pos];
        mat[current_pos] = swap_val;
      }
  }
```
Value State Dependence Graph

- Enabling outer loop pipelining:
  - Now outer loop can also be flattened like the inner loop.
Introduction to Spatial Computation

extern int a[], b[];
void g(int *p, int i)
{
    b[i+2] = i & 0xf;
    a[i] = b[i] + *p;
}

(A)

(B)

Static Dataflow Graph in Hardware (SPDE)
<table>
<thead>
<tr>
<th></th>
<th>Superscalar</th>
<th>Custom Computation</th>
<th>Spatial Computation</th>
<th>Spatial Computation (+ VSDG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control-Flow Speculation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dataflow Execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic Execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple Flows of Control</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic Memory Level Parallelism</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Custom Computation

• Does not help with speed-up limits due to Amdahl's law
  – Poor sequential code performance can make it worse.

• Helps with speed-up limits due to Utilization Wall
  – Due to very high energy efficiency, parallel code can run faster

• Does not help with programmability
  – makes it worse!
Spatial Computation

- Does not help with speed-up limits due to Amdahl's law
  - Poor sequential code performance can make it worse.
- Helps with speed-up limits due to Utilization Wall
  - Due to very high energy efficiency, parallel code can run faster
- Helps with Programmability
  - Suitable for Von Neumann Computation model
# Brief History of Computer Architecture

<table>
<thead>
<tr>
<th>ILP</th>
<th>TLP</th>
<th>DLP</th>
<th>Aggressive Superscalar</th>
<th>Multicore / Manycore</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>~</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>?</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+</td>
</tr>
</tbody>
</table>

- **Energy Efficiency**
  - Aggressive Superscalar: Very Poor
  - Multicore / Manycore: Poor

- **Programming Complexity**
  - Aggressive Superscalar: Low
  - Multicore / Manycore: High

- **Hardware Design Complexity & Scalability**
  - Aggressive Superscalar: Very Poor
  - Multicore / Manycore: Good
## Recap

<table>
<thead>
<tr>
<th></th>
<th>Aggressive Superscalar</th>
<th>Multicore / Manycore</th>
<th>Custom Computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td>~</td>
<td>?</td>
<td>-</td>
</tr>
<tr>
<td>TLP</td>
<td>?</td>
<td>+</td>
<td>~</td>
</tr>
<tr>
<td>DLP</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Energy Efficiency</td>
<td>Very Poor</td>
<td>Poor</td>
<td>Very Good</td>
</tr>
<tr>
<td>Programming Complexity</td>
<td>Low</td>
<td>High</td>
<td>Very High</td>
</tr>
<tr>
<td>Hardware Design Complexity &amp; Scalability</td>
<td>Very Poor</td>
<td>Very Good</td>
<td>Very Good</td>
</tr>
</tbody>
</table>
# Brief History of Computer Architecture

<table>
<thead>
<tr>
<th>ILP</th>
<th>TLP</th>
<th>DLP</th>
<th>Aggressive Superscalar</th>
<th>Multicore / Manycore</th>
<th>Conventional Custom Computing</th>
<th>Spatial Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="ILP" /></td>
<td><img src="image" alt="TLP" /></td>
<td><img src="image" alt="DLP" /></td>
<td><img src="image" alt="Aggressive" /></td>
<td><img src="image" alt="Multicore" /></td>
<td><img src="image" alt="Conventional" /></td>
<td><img src="image" alt="Spatial" /></td>
</tr>
<tr>
<td><img src="image" alt="Energy Efficiency" /></td>
<td>Very Poor</td>
<td>Poor</td>
<td>Very Good</td>
<td>Very Good</td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Programming Complexity" /></td>
<td>Low</td>
<td>High</td>
<td>Very High</td>
<td>Low ??</td>
<td></td>
<td></td>
</tr>
<tr>
<td><img src="image" alt="Hardware Design Complexity &amp; Scalability" /></td>
<td>Very Poor</td>
<td>Very Good</td>
<td>Good</td>
<td>Good</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>