Practical Reconfigurable Computing Today

NCSA Reconfigurable Systems Summer Institute
July, 2005

Michael Babst
DSPlogic, Inc.
www.dsplogic.com
301-977-5970 x705
Typical Design flow (the old way)

- Benchmark performance, profile execution, I/O
- Partition Algorithm
- Define CPU/FPGA messaging scheme
- Learn VHDL or Verilog
- Design FPGA
  - Code Application I/O (interface w/ custom vendor cores)
  - Verify I/O - Synthesize/Place/Route FPGA
  - Optimize I/O for BW/latency
  - Code Application Core (verify)
  - Synthesize/Place/Route FPGA (App+I/O)
  - Fiddle with I/O, Application until FPGA builds
  - Verify APP+I/O - Synthesize/Place/Route FPGA
  - Done!
  - Make a small change to the application
  - Become an Expert in VHDL/Verilog, HW design, timing diagrams...
  - Redesign I/O, Modularize design, Isolate I/O from application
  - Optimize speed and timing
- Working design!
Typical Design flow (the old way)

- Here comes the Virtex 4!
- Re-partition Algorithm
- Redesign I/O ...
- Get the idea?

RC can be a challenge, even for a hardware designer
Customers #1 Requirement in selecting (or not selecting) an RC platform

- **Usability of Tools**

Currently available methods
- High-level C programming
  - Mitrion-C, System-C, Handel-C
- Hard-core HDL
  - VHDL, Verilog
- Model-based / Dataflow Design
  - Viva, Simulink

Customers need a solution that...
1) works today
2) is easy to use.
Agenda

- Rapid RC Development Kit
- Reconfigurable Computing (RCIO) API
- RCIO API Implementation
  - RCIO FPGA Core / SW Library
  - Cray XD1 Platform
- Application Examples
  - FFT
  - Point Projection
  - Floating point Multiply/add/subtract
Rapid RC Development Kit
Key Development Kit Components

- RCIO Software Library
  - RCIO API Implementation, platform optimized
- RCIO FPGA Core
  - RCIO Core Implementation, platform optimized
- Matlab/Simulink Interface
  - DSPlogic RC Blockset for Matlab/Simulink
  - RCIO FPGA Builder
  - Automated FPGA implementation, platform optimized
  - Reliable one-click build process
  - Xilinx System Generator Application Example
Rapid RC Development Kit – Design Flow

- Rapid implementation
- Simplified CPU/FPGA messaging
  - Low-overhead
  - Bandwidth or latency optimization
  - Stream or block processing
- Common API
  - Portable, reusable upgradeable code
- Optimized core libraries
  - Efficient resource utilization
- Integrated algorithm verification
- Integrated bitstream generation
- Simplified high-speed design
  - Maximum processing and I/O throughput
- Floating-point capability

Implement Algorithm
Call RCIO API Functions

Algorithm
CPU/FPGA Partition, Specify Dataflow
DSPlogic RC Blockset

Simulink
Familiar, industry standard modeling environment

Matlab/Simulink

Verify Data Processor Output
DSPlogic RCIO FPGA Builder

Design Data Processor

Fully Integrated, Verified, Seamless Application
RCIO API Transparent interface

Processor

DSPlogic
Model-based FPGA Design

Advantages

- VHDL not required (but possible)
- Integrated algorithm verification
- Industry standard, familiar environment
- Easy integration of IP cores
- Clear view of algorithm architecture
- Optimized core libraries
  - Highly efficient use of FPGA resources
- Ease of FPGA design verification
- Automatic design documentation
- Integrated bitstream generation

Compatible with high-level languages

- HLL Behavioral descriptions possible
DSPlogic/Xilinx System Generator Integrated Environment

Direct Algorithm To Platform FPGA!
Reconfigurable Computing I/O (RCIO) API
Reconfigurable Computing I/O API

- A Simple, future-proof CPU/FPGA messaging interface

- Dramatically reduces FPGA development time
- Quickly achieve optimum latency and bandwidth
  - Crucial to performance
- Application portability, reusability and upgradeability
- Future-proof: easy migration to newer, higher-performance FPGAs
- Separate data and control message paths
- Low-overhead
- Block and Stream Processing
- Multiple CPU/FPGA support

Diagram:
- User Software Application
- RCIO Library and FPGA Core
- User FPGA Application
- Transparent portable interface
- rcio_send()
- rcio_receive()
- rcio_appcfg()
- rcio_appstat()
Reconfigurable Computing I/O API

- Multiple processor / multiple FPGA support

CPU #1
- User Application
- RCIO SW library

CPU #2
- User Application
- RCIO SW library

CPU #N
- User Application
- RCIO SW library

FPGA #1
- RCIO FPGA Core
- User Application
- FPGA #1

FPGA #2
- RCIO FPGA Core
- User Application
- FPGA #2

FPGA #K
- RCIO FPGA Core
- User Application
- FPGA #K

Platform Specific Connection Fabric

Transparent, Portable Application Interface!

MPI, PVM, etc
Data Message Structure (64-bit)

- **Dataset**
  - MSG\(_0\)
  - MSG\(_1\)
  - MSG\(_M-1\)
  - M Messages

- **Message**
  - BLK\(_0\)
  - BLK\(_1\)
  - BLK\(_B-1\)
  - B Blocks

- **Data Block**
  - word\(_0[63:0]\)
  - word\(_1[63:0]\)
  - word\(_K-1[63:0]\)
  - K words

- **User-definable format**
  - real32
  - real32
  - real64
  - int16
  - int16
  - int16
  - int16
  - custom

**I/O often limits performance**
- Use care with CPU/FPGA algorithm partitioning
- Consider smaller data types

**Easily group processing blocks into messages for optimal I/O bandwidth**
Software API - Data Interface

- `rcio_send()`: Send single message to FPGA
- `rcio_receive()`: Receive single message from FPGA
- `rcio_stream()`: Blocking function
  - Break dataset into messages
  - Send all messages to FPGA
  - Receive all result messages from FPGA
- Bandwidth-optimized
Software API - Control Interface

- **Messaging control functions**
  - `rcio_config()`
    - Initialize / configure CPU/FPGA communications link
  - `rcio_status()`
    - Return status of communications link
      - `fpgaStatus`
      - `nMsgReceived, nMsgReturned`
      - Fifo levels / over/underflow
  - `rcio_close()`
    - Close CPU/FPGA communications link

- **User application control and status commands**
  - `rcio_appcfg()`
    - Write to user-definable application control register
  - `rcio_appstat()`
    - Read user-definable application control register
RCIO Hardware Abstraction Layer (HAL) API

DSPlogic™

RCIO Core

Input Data Message FIFO

Output Data Message FIFO

Control I/O

User FPGA Application

Data Processor

Platform Specific RAM / Peripherals

in_ready
in_data
in_write
in_start
in_length
out_ready
out_data
out_write
out_start
out_length
ctrl_reg(0-7)
stat_reg(0-7)
ib_depth, ob_depth

clk
rst
RCIO Hardware Abstraction Layer (HAL) API

- FPGA design tool-independent
- Supports wrappers in all design environments
  - High-level design tools
    - Xilinx System Generator, etc.
  - Custom VHDL / Verilog
  - High-level C
    - Mitrion-C System C, Handel-C, System Verilog, etc.
RCIO FPGA Core / SW Library
Implementation for the Cray XD1 Supercomputer
DSPlogic™ RCIO SW Library
rcio_send(fpga_id, *datap, txMsgLen)

Seamless Cray XD1™ CPU-FPGA Messaging

Directly Link CPU and FPGA Applications!

Total Hardware Abstraction!

DSPLlogic™ RCIO FPGA Core
ready
data
write
start
length

Application Processing

RAP

Processor
Application Memory
**Cray XD1 Performance**

- Demonstrated performance and usability on multiple applications in multiple design environments
- **Extremely Modular - Multiple applications at full speed (200 MHz)**

**Fastest CPU/FPGA Interface Available for the Cray XD1!**

**Achievable Data Rates (Mbytes/sec) (Including dataset sizes > 2 MB)**

<table>
<thead>
<tr>
<th></th>
<th>Send</th>
<th>Rcv</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theoretical Max (not achievable)</td>
<td>1422</td>
<td>1422</td>
<td>3022</td>
</tr>
<tr>
<td>Typical Send Only Application</td>
<td>1100</td>
<td>N/A</td>
<td>1100</td>
</tr>
<tr>
<td>Typical Receive Only Application</td>
<td>N/A</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>Typical Send/Receive Application</td>
<td>&lt;1100</td>
<td>&lt;1100</td>
<td>1100</td>
</tr>
<tr>
<td>**DSPlogic RCIO Send/Receive ***</td>
<td>825</td>
<td>825</td>
<td>&gt;1650</td>
</tr>
</tbody>
</table>

* Using rcio_stream()
Application Examples
- FFT
- VHDL-based design flow
FFT Accelerator

- FFT Length: 32 to 65536
- Fixed-point, full-precision
- Complex FFT
- VHDL design flow
- Device Utilization
  - 14k Slices (60% V2P50)
  - 186 Block Rams (80% V2P50)
- Software API
  - Include RCIO API
  - Additional application specific library functions
    - fft_init(fft length, direction)
  - FFT Usage
    - rcio_config()
    - fft_init()
    - rcio_stream()
    - rcio_close()
FFT Performance Improvement

- ~10x improvement possible today!
  - Complex FFT, FPGA vs. FFTW on AMD 246
- Performance depends on data types

<table>
<thead>
<tr>
<th>Im₁(15:0)</th>
<th>Re₁(15:0)</th>
<th>Im₀(15:0)</th>
<th>Re₀(15:0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>unused</td>
<td>Re₀(15:0)</td>
<td>unused</td>
</tr>
<tr>
<td>Im₁(15:0)</td>
<td>Re₁(15:0)</td>
<td>Im₀(15:0)</td>
<td>Re₀(15:0)</td>
</tr>
<tr>
<td>out</td>
<td>Im[31:0]</td>
<td>Re[31:0]</td>
<td></td>
</tr>
</tbody>
</table>

Additional speed at expense of considering scaling / dynamic range effects

Accuracy similar to single-precision floating point algorithms
FFT Summary

- Performance is I/O constrained
- > 10x speed gains are achievable today
- FPGA Performance enhancement increases with FFT length
- Multiple FFTs utilize pipeline and provide efficiency
- FFT L2norm accuracy \( \sim 10^{-5} \), similar to other single-precision algorithms
- Modular architecture
  - Separate I/O and application optimization
  - Rapid application development
- Message latency limits speed improvement for single computations of small FFT sizes
Application Examples

-Dirt Code - Point Projection
-Model-based design example
struct s_point pointprojection(struct s_plane plane, struct s_point p1)
{
    double temp;
    struct s_point vec1, proj;
    vec1.x = p1.x - plane.p.x;
    vec1.y = p1.y - plane.p.y;
    vec1.z = p1.z - plane.p.z;
    temp = vec1.dot(plane.n);
    proj.x = vec1.x - temp*plane.n.x + plane.p.x;
    proj.y = vec1.y - temp*plane.n.y + plane.p.y;
    proj.z = vec1.z - temp*plane.n.z + plane.p.z;
    return proj;
}

Courtesy David Raila / Youssef Hashash
struct s_point

pointprojection(struct s_plane plane, struct s_point p1)

vec1.x = p1.x - plane.p.x;
vec1.y = p1.y - plane.p.y;
vec1.z = p1.z - plane.p.z;

temp = plane.n.x * vec1.x +
      plane.n.y * vec1.y +
      plane.n.z * vec1.z;

proj.x =
         vec1.x - temp * plane.n.x + plane.p.x;
proj.y =
         vec1.y - temp * plane.n.y + plane.p.y;
proj.z =
         vec1.z - temp * plane.n.z + plane.p.z;

return proj;
}
Message Formats

- **Input Format**

<table>
<thead>
<tr>
<th>Block</th>
<th>64-bit offset</th>
<th>in_data[63:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>pt[0].x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>pt[0].y</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>pt[0].z</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>plane_n[0].x</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>plane_n[0].y</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>plane_n[0].z</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>plane_p[0].x</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>plane_p[0].y</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>plane_p[0].z</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>pt[1].x</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>pt[1].y</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>pt[1].z</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>plane_n[1].x</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>plane_n[1].y</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>plane_n[1].z</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>plane_p[1].x</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>plane_p[1].y</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>plane_p[1].z</td>
</tr>
</tbody>
</table>

- **Output Format**

<table>
<thead>
<tr>
<th>Block</th>
<th>64-bit offset</th>
<th>out_data[63:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>proj[0].x</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>proj[0].y</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>proj[0].z</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>proj[1].x</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>proj[1].y</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>proj[1].z</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block P-1</td>
<td></td>
<td>proj[P-1].x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>proj[P-1].y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>proj[P-1].z</td>
</tr>
</tbody>
</table>

- **Number of Projections/Message**
  - P = 64*k, 6 <=k <= 113

- **Theoretical Max possible projections / second**
  - I/O limited

- **No Packing**
  - ~ 200 MHz/9 = 22.2 M projections/sec

- **With Packing**
  - ~ 200 MHz/3 = 66.6 M projections/sec
Data Format / Precision

- **Data Width**
  - Use 64-bit for future flexibility
  - Tradeoff data packing vs. software/firmware rework

  **Input Data**
  - din[15:0]

  **Output Data**
  - dout[63:0]

- **Precision**
  - Input: 16-bit signed integer
  - Output: 52-bit signed integer (sign-extended to 64-bits)
vec1.x = p1.x - plane.p.x;
vec1.y = p1.y - plane.p.y;
vec1.z = p1.z - plane.p.z;

temp = plane.n.x*vec1.x + plane.n.y*vec1.y + plane.n.z*vec1.z;

proj.x = vec1.x - temp*plane.n.x + plane.p.x;
proj.y = vec1.y - temp*plane.n.y + plane.p.y;
proj.z = vec1.z - temp*plane.n.z + plane.p.z;
Performance results

- **Computation Rate**
  - (conservative)

- **Input data type precision**
  - ~$10^{-5}$

- **Computation Accuracy**
  - Full-precision
  - L2Norm Error = 0

- **4x additional speed improvement (60M projections/sec) with data packing**

- **Next steps**
  - Move more functionality into FPGA
  - Partition algorithm for lower I/O bandwidth
Summary

- **Rapid RC Development Kit**
  - A practical design method available today
- **Reconfigurable Computing (RCIO) API**
  - Offers many benefits, including portability
- **RCIO API Implementation**
  - Cray XD1 Platform – fastest FPGA/CPU interface available
  - Additional platforms coming
- **Application Examples**
  - FFT
  - Point Projection
  - Floating point multiply/add/subtract
- **Future**
  - Hardware co-simulation
  - CPU/FPGA I/O Standardization
  - RCIO API enhancements
  - High-level language integration
Contact Information

DSPlologic

Michael Babst
www.dsplologic.com
301-977-5970 x705
mike.babst@dsplologic.com