Domain-Specific Programming of Very High Speed Packet Processing

Gordon Brebner
Xilinx Labs
San José, USA
Programmable logic, 1989

- Circuit design tools used
- Can only implement simple functions

- Computer science research perspective:
  - Gates and wiring to be programmed
  - Scarce resource to be managed
  - Limited to niche applications

Cell: 2-input gate
Local wiring between cells
Both programmable by writing to memory

1000 cells

© Copyright 2012 Xilinx
Programmable logic, 2012

- Hardware design tools used
- Can implement complex systems

Computer science research perspective:
- Software-style engineering
- Adaptable processing architectures
- Lower-power peer of CPU, GPU, NPU

Need theory

Cell: 6-input gate and 2 flip-flops
Embedded function blocks and memories
Local and longer wiring between components
All programmable by writing to memory

2m cells
Programming programmable logic

- Chip design experience
- Hardware Description Language (HDL)
- Cryptic results
  - Behaviour and performance

- Enhancements (for ‘hardware guys’):
  - Libraries of blocks
    - Allow re-use and sharing
    - In HDL, or pre-synthesised
  - High-level synthesis
    - Usually superset of subset of C
    - Translated into HDL

- Abstraction is lacking
The packet processing domain

The Line Card

- **Data Framing**: Represent packet, and compute check code
- **Packet Classification**: Extract key from packet, then look key up in table
- **Packet Editing**: Modify fields in packet, or restructure packet
- **Traffic Management**: Queue packet in memory, and schedule departure

© Copyright 2012 Xilinx
Programming packet processing

<table>
<thead>
<tr>
<th>Component</th>
<th>Speed requirement</th>
<th>Programmability requirement</th>
<th>Typical NPU feature</th>
<th>Programmable logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classification</td>
<td>High</td>
<td>Medium: parser only</td>
<td>Specialised multi-threaded processors</td>
<td>Programmable classifier</td>
</tr>
<tr>
<td>Editing (slow path)</td>
<td>Medium</td>
<td>High</td>
<td>General-purpose processors + assists</td>
<td>Soft multicore, plus accelerators</td>
</tr>
<tr>
<td>Editing (fast path)</td>
<td>High</td>
<td>Medium</td>
<td>Highly multi-threaded processors + assists</td>
<td>Programmable parser/editor</td>
</tr>
<tr>
<td>Traffic management</td>
<td>High</td>
<td>Low: scheduler only</td>
<td>Hardware accelerator</td>
<td>Configurable traffic manager</td>
</tr>
</tbody>
</table>

Basis for a domain-specific language?
First Generation
Packet processing data paths  1/2G  Xilinx Virtex-II Pro

- Centred around Click (MIT 2001)
  - Designed for building modular routers from components
  - Has software implementation in C++

- Systems have data flow model
  - *Elements* process packets
  - *Connections* to move packets between elements

- Looked promising as a model for hardware systems
Each box is an instance of a pre-defined Click element.

Connections between elements form a data flow graph.

Some characteristics:
- Fixed-function elements
- Fine-grain element functions
- Inter-element interaction:
  - data flow for packets
  - method calls otherwise
Cliff: Click for FPGA

- Processed Click in its textual form
- Generated Verilog description of design

- Elements implemented in Verilog
  - Finite State Machine
  - Two predefined states
    - Receive packet
    - Transmit packet
  - User defined states in between
    - Packet handling
    - Accessing memory

- Connections between elements
  - Three-way handshake protocol
  - Bus widths up to 344 bits
## Cliff results for three benchmarks

<table>
<thead>
<tr>
<th>Application</th>
<th>Area (slices)</th>
<th>Frequency (MHz)</th>
<th>Throughput (Gb/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPv4 router</td>
<td>4016</td>
<td>125</td>
<td>2</td>
</tr>
<tr>
<td>NAT</td>
<td>3248</td>
<td>125</td>
<td>2</td>
</tr>
<tr>
<td>DiffServ</td>
<td>9114</td>
<td>85</td>
<td>1.6</td>
</tr>
</tbody>
</table>
Lessons learnt
- Click elements very fine-grain
  - Inefficient in hardware: 90% of resource used for interface
- Click elements had to be implemented in Verilog by hand

Conclusions
- Click not attractive as the domain-specific language choice
- Has potential for connecting coarser-grain elements

Result
- Click semantics generalised to add non-packet type connections
- Spun out as horizontal technology for general system building
  - And used in next generation work
Second Generation
Packet processing pipeline 10/20G Xilinx Virtex-4/5

-Originated in a collaboration with Bell Labs

-Packet processing tool suite
  - Click used for system building
    • Describe system in terms of connected components and subsystems
  - Invented G
    • New packet-centric and protocol-agnostic language for creating components
    • Initial focus on packet editing functions

-G compiled to synthesisable HDL
  - Describing a customised pipeline architecture instance
Summary of main G constructs

<table>
<thead>
<tr>
<th>Test Values</th>
<th>Change Values</th>
<th>Insert New Fields</th>
<th>Remove Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Packet</td>
<td>Packet</td>
<td>Packet</td>
<td>Packet</td>
</tr>
<tr>
<td>Payload</td>
<td>IP Header</td>
<td>IP Header</td>
<td>IP Header</td>
</tr>
<tr>
<td></td>
<td>Type</td>
<td>Type</td>
<td>Type</td>
</tr>
<tr>
<td></td>
<td>Control</td>
<td>Control</td>
<td>Control</td>
</tr>
<tr>
<td>Destination Address</td>
<td>Destination Address</td>
<td>Destination Address</td>
<td>Destination Address</td>
</tr>
</tbody>
</table>

```
[packet.type == 0x8100] {
  // conditional operations
}
```

```
set packet.SA = 0x010203040506
```

```
set TC.Type= 0x8100;
set TC.Control = 0x123456;
insert TC after TC1;
```

```
remove TC1;
remove TC2;
```

- G is a declarative language
- Rules can have conditional guards
- Concurrent execution of rules is the default semantics
- G is not general purpose, but is Turing complete
- No implementation detail included: “what” not “how”
Examples of G format declarations

// MPLS header format
format F_MPLS_header = {
    MPLS_LABEL : 20,
    MPLS_COS : 3,
    MPLS_S : 1,
    MPLS_TTL : 8
};

// IPv4 header format
format F_IPv4_header = {
    IPv4_VERSION : 4,
    IPv4_IHL : 4,
    IPv4_COS : 8,
    IPv4_ID : 16,
    IPv4_FLAGS : 3,
    IPv4_FRAG_OFFSET : 13,
    IPv4_TTL : 8,
    IPv4_PROTOCOL : 8,
    IPv4_HDR_CHKS : 16,
    IPv4_SA : 32,
    IPv4_DA : 32
};

// Input and output packet format
format F_packet = {
    : 16,               // unused
    type : 16,          // type code
    { : ( MPLS : F_MPLS_header,       // >=1 label case
          { IPv4next : F_IPv4_header | // 1 label subcase
            MPLSnext : F_MPLS_header // >1 label subcase
          }
        )
    } |
    IPv4 : F_IPv4_header    // no label case
},
    : *                    // rest of packet
};
Example G handling rules: for MPLS

```c
[ type == C_MPLS_TYPE_CODE ] {  // MPLS packet  // Forward packet if still alive  set oTTL = MPLS.MPLS_TTL - 1;  [oTTL > 0] forward /*F_packet*/ on packetout;

  // Obtain routing operation from lookup on MPLS label  read MPLS_result from MPLS_lookup [ MPLS.MPLS_LABEL ];

  [ MPLS_result.type == C_SWAP_TYPE_CODE ] {  // Still have one or more MPLS labels  remove MPLS;
    set MPLS.MPLS_TTL = oTTL;
  }

  [ MPLS_result.type == C_PUSH_1_TYPE_CODE ] {  // Still have one or more MPLS labels  set shim1.MPLS_LABEL = MPLS_result.label1;  set shim1.MPLS_COS = MPLS.MPLS_COS;  set shim1.MPLS_S = 0;  set shim1.MPLS_TTL = oTTL;
    insert shim1 after type;
  }

  [ MPLS_result.type == C_PUSH_2_TYPE_CODE ] {  // Still have one or more MPLS labels  set shim1.MPLS_LABEL = MPLS_result.label2;  set shim1.MPLS_COS = MPLS.MPLS_COS;  set shim1.MPLS_S = 0;  set shim1.MPLS_TTL = 255;
    set shim2.MPLS_COS = MPLS.MPLS_COS;  set shim2.MPLS_S = 0;  set shim2.MPLS_TTL = 255;
    insert shim1, shim2 after type;
  }

  [ MPLS_result.type == C_PUSH_3_TYPE_CODE ] {  // Still have one or more MPLS labels  set shim1.MPLS_LABEL = MPLS_result.label3;  set shim1.MPLS_COS = MPLS.MPLS_COS;  set shim1.MPLS_S = 0;  set shim1.MPLS_TTL = oTTL;
    set shim2.MPLS_LABEL = MPLS_result.label2;
    set shim2.MPLS_COS = MPLS.MPLS_COS;
    set shim2.MPLS_S = 0;
    set shim2.MPLS_TTL = 255;
    set shim3.MPLS_LABEL = MPLS_result.label1;
    set shim3.MPLS_COS = MPLS.MPLS_COS;
    set shim3.MPLS_S = 0;
    set shim3.MPLS_TTL = 255;
    insert shim1, shim2, shim3 after type;
  }

  [ MPLS_result.type == C_SWAP_PUSH_TYPE_CODE ] {  // Still have one or more MPLS labels  set MPLS.MPLS_LABEL = MPLS_result.label1;
    set MPLS.MPLS_TTL = oTTL;
    set shim1.MPLS_LABEL = MPLS_result.label2;
    set shim1.MPLS_COS = MPLS.MPLS_COS;
    set shim1.MPLS_S = 0;
    set shim1.MPLS_TTL = 255;
    insert shim1 after type;
  }

  [ MPLS_result.type == C_IPv4_TYPE_CODE ] {  // Packet will be forwarded as IPv4 instead of MPLS  set type = C_IPv4_TYPE_CODE;
    set IPv4next.IPv4_TTL = oTTL;
  }
}
```
Implementation detail is separate
Soft architecture generation from G

- Standard G compiler generates a ‘best effort’ pipeline architecture (described in HDL) that fits the specific G program

- Can do automated architecture solution space exploration for throughput, latency, resource trade-offs

Slices (resource unit) vs. latency

- The two main clusters are the lower latency, and lower resource, solutions
- The ‘best’ solutions are those at the lower-left part of each ellipse

Elipses denote solutions with same throughput
Multi-level packet-centric testing/debugging

Input packet set description → Packet Generator → Packet Translator → Hardware in the loop → Packet Translator

Click/G-level Simulation → Packet Translator → RTL level Simulation → Packet Translator

“Golden” output packet set description

Equivalent?

© Copyright 2012 Xilinx
Example ‘out of the box’ results

<table>
<thead>
<tr>
<th>Data width</th>
<th>Max. throughput (Gbit/sec)</th>
<th>Latency (nsec)</th>
<th>Virtex-5 slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit</td>
<td>6.05</td>
<td>63</td>
<td>762</td>
</tr>
<tr>
<td>64-bit</td>
<td>11.90</td>
<td>54</td>
<td>993</td>
</tr>
<tr>
<td>128-bit</td>
<td>23.42</td>
<td>49</td>
<td>1379</td>
</tr>
<tr>
<td>256-bit</td>
<td>45.06</td>
<td>51</td>
<td>1969</td>
</tr>
</tbody>
</table>

Smallest Virtex-5 device (LX30) has 4800 slices

**MPLS label switching router:**

G code written, simulated, and analysed within two days
Provided as a ‘boutique’ tool set to selected top tier customers
  – Used on real product design projects
  – Did not become official software product – too domain-specific
  – Also made available for NetFPGA platform

G defined in 2005, and extended in 2007

Fast G compiler, but then slow back-end FPGA implementation flow

G somewhat sub-domain-specific: emphasis on packet editing
  – Did not have enough generality in packet parsing

H – extended version of G – was defined in 2006
  – Architecture developed in 2008, but compiler never implemented
  – Ideas carried forward into next generation work
Initial focus on Packet Parsing only
- Allow header-by-header analysis
- Extract header fields as keys for classification lookup
- Skip over outer headers to reach inner contents

Desire for soft programmability at run time
- Without resynthesising the hardware

Desire for unthreatening programming language
- Less declarative, more imperative
## PP (Packet Parsing) language overview

### PP description:

<table>
<thead>
<tr>
<th>Parser directives</th>
</tr>
</thead>
<tbody>
<tr>
<td>throughput, parse depth, initialisation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>constants and common references</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Header class definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>header format, header parsing rules, key building rules</td>
</tr>
</tbody>
</table>

Five methods:
- next_header
- next_offset
- key_builder
- earliest, latest

### PP execution model:

```plaintext
while true do {
    input packet;
    header := first header;
    while not done do {
        apply rules for header;
        header := next header;
    }
    output packet and results;
}
```

- Use *one* high-level program to
  - Optimise hardware implementation
  - Enable run time modification and on-the-fly updates
### PP example class

```plaintext
class MPLS_TYPE {
    struct { label : 20,
             cos  : 3,
             sBit : 1,
             ttl  : 8 }

    method next_header =
        if (sBit == 0){
            MPLS_TYPE;
        } else {
            ETH_TYPE;
        }

    method next_offset = size();
    method key_builder = {label}
    method earliest = 1;
    method latest = 3;
}
```

- **Structure indicates header format**
- **Next header method indicates what the next header to parse will be**
- **Next offset method indicates where the next header to parse will be found**
- **Key builder method constructs context**
- **Earliest and latest methods indicate bounds on the header’s location in the protocol stack**
PP compiled to programmable pipeline

Microarchitecture instance
Micro-programmed features

- Parsing requirements: multiple protocols possible per stage
  - Operand setup
    • Fields to extract: location and sizing
    • Constants used: value and sizing
  - Operations performed
    • Operands used: selection between fields, constants, intermediate results

- Tailored microcode stored locally in each stage
  - One microcode entry per protocol that can be parsed in stage
  - Microcode’s structure and size depend on parsing requirements

<table>
<thead>
<tr>
<th>Protocol microcode entry</th>
<th>Extract Size</th>
<th>Extract Offset</th>
<th>• • •</th>
<th>Compute Operation</th>
<th>Compute Input</th>
<th>• • •</th>
<th>Key Build Operation</th>
<th>Key Source</th>
<th>Source Size</th>
<th>• • •</th>
<th>Constant</th>
<th>• • •</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fields to extract from data stream</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method operations to perform</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key building operations to perform</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constants to use</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Throughput versus FPGA utilisation

- Trade-offs obtained by varying data path width
Status

▸ PP work is basis for Xilinx 100G Packet Processing product
  – First release in October 2012

▸ Architecture being extended beyond parsing pipeline
  – Incorporate lookup engines
  – Incorporate custom engines (e.g. checksum) and user engines

▸ PP language extension for second release
  – Programmable packet editing – incorporate G and H material
  – Integration of parsing, lookup, and editing
Central challenge is very wide data paths
- 1K, 2K, 4K bits wide
- Multiple packets can reside in one word

Idealised world for multi-packet parallelism
- Fixed mapping of packets to data path words
  - Just handle different segments of words in parallel

Real world issues:
- Packets have variable length
- Packets may have different stacks of headers
- Packet headers may have different formats
Invention of parallel packet extraction technology (2011)

Localise regions of highway to particular packet handlers
  – Approximation to the idealised world of a segmented highway
  – Scalable approach using fixed-width parallel pipelines

Bleeding-edge prototypes being used in “Virtex-8” planning
  – Evaluate silicon architecture options

Target for PP without dramatic language changes
Conclusion

Programmable logic has good impedance match with networking
  – Delivers high flexibility and high performance packet processing
    • Via soft parallel and pipelined architectures

High-level programming is promising for well-matched domains
  – Four research generations show good results for packet processing
    • Still converging on exact notion of what functions the domain requires

Hardware engineers have got the physical technology there over the past 25+ years …

… and now is a good time for software engineers to take a serious interest in harnessing the technology