Managing Complex Trace Filtering and Triggering Capabilities of CoreSight™

Jens Braunes
pls Development Tools
Outline

- Benefits and challenges of on-chip trace
  - The evolution of embedded systems and the requirements on debugging
- The CoreSight architecture and programming model
  - A standardized approach for high system visibility
- Managing the programming model
  - The requirements to the debug tools
- Architecture description for debugger adaptation
  - Provide all needed information to adapt the debugger
- Conclusion
Benefits and Challenges of On-Chip Trace
Evolution of Embedded Systems

- Embedded systems in the past:
  - 1 processor
  - 1 bus
  - Low clock rates
  - Etc.
  - Debugging was reasonable simple

- Embedded systems today and in future:
  - Many processors
  - Many busses
  - High clock rates
  - Etc.
  - Debugging getting more and more complex
Two Worlds of Debugging

- Traditional debugging
  - No real world execution possible
  - Execution externally controlled (debugger)
  - Typical tools: breakpoints, watchpoints, single step execution
  - Invasive - core halted

- Tracing
  - Non-invasive - core runs at full speed
  - Collect executed instructions / data transfers
  - Traced data delivered off-chip in real-time or stored in on-chip trace buffer
On-Chip Trace

- Limitations of trace:
  - Trace port with limited bandwidth
  - High pin costs – low pin count required
  - Trace memory with limited capacity

- Only relevant data has to be captured
- Paradigm shift from post-trace analysis to pre-trace specification

- User has to specify
  - What has to be captured?
  - When it has to be captured?
On-Chip Trace (cont)

On-Chip trace preprocessing requires extensive HW support:

- On-chip filter capabilities
- Trace data compression
- Trigger logic
- Cross triggers (multi-core / multi-component)

- Savings of pin costs outbalance area consumption

- **CoreSight is ARMs on-chip trace solution for future SoCs and multi-core systems**
The CoreSight™ Architecture and Programming Model
CoreSight™ On-Chip Debug Architecture

For details:
ARM; CoreSight™ Components Technical Reference Manual

ARM Developers’ Conference & Design Pavilion 2007
CoreSight™ Features and Capabilities

- Debug Access Port (DAP) provides real-time access to JTAG scan chains, AMBA busses, debug / trace registers
- Embedded Trace Macrocell (ETM):
  - Non-intrusive observation of core internal busses (instructions and data)
  - Event driven trace capture
  - Powerful user configurable triggers and filters
- Embedded Cross Trigger (ECT) distributes debug events between cores or ETMs
- AHB Trace Macrocells (HTM) allows observation of bus operations, not visible by ETMs
- Trace funnel combines multiple trace sources into a single trace stream
- Embedded Trace Buffer (ETB) for on-chip storage of captured trace data
CoreSight Programming Model

- Everything is hierarchical
  - Overall system is build from components (ETMs, HTMs, Embedded Cross Trigger, ...)

- Register based programming model for each CoreSight component
  - Registers for identification and management
  - Component specific control registers

- Memory mapped interface to provide access to component registers

- Typically access via AMBA 3 APB
Managing the Programming Model
Example: Programming the ETM

- Controlling tracing ➔ program the control registers
- Resources for filtering and triggering
  - Comparators (addresses, address ranges, data, context ID)
  - Counters, sequencer
  - External inputs

From the users perspective: Which functionality is behind it?

Example
Enable trace recording
Example: Programming the ECT

- Routing of events between CoreSight components via channels
- Programmable connections of external inputs/outputs from/to channels

From the users perspective: Which functionality is behind it?
Managing Complex Programming Model

- Large number of registers to program
- Fundamental trace functionality mapped to more than one register
- E.g. start trace at address A and stop trace at address B:
  1. Set address comparators to recognize A and B
  2. Set register TraceEnableEvent
     - To always enable tracing
  3. Set register TraceEnableCtrl1
     - To activate trace start/stop logic
     - To ignore all include/exclude addresses and address ranges
  4. Set register TraceStartStop
     - To start trace at address A
     - To stop trace at address B
Managing Complex Programming Model

- User do not like to know details about the internals
- User has a certain debug task in mind
- Complexity have to be hidden by debugger

Two possibilities:

1. Dialog based:
   - All configurations set up by using fixed dialog boxes
   - 'Hard-coded' functionality
   - Customizable by using parameters
   - Inflexible in terms of target configuration

2. Language-based:
   - Trigger and filter setup by using a definition language
   - Functionality can be build from atomic language elements
   - Flexible in terms of target configuration
Language-based Debug Configuration

- Realized in the Universal Emulator Configurator

Adaptable to different on-chip emulators

Graphical Trace Configuration

Hardware Independent Configuration Language

Hardware Dependent Configuration Language

Configuration Data

- User-friendly configuration of on-chip emulators
- Hardware realization transparent to the user, knowledge about real debug hardware not necessary
- Nevertheless experts keep full control

But:
Different modules at lowest level for different on-chip debug architectures
Full Adaptation of Debug Tool

- What is missing here?
  - Definition of trace functionality
  - Mapping of trace functions to HW resources

Architecture Specification

Mechanism to adapt the debugger

ARM Developers’ Conference & Design Pavilion 2007

(C) pls Programmierbare Logik & Systeme GmbH
Architecture Description for Debugger Adaptation
HW Dependent Configuration Language

- Adaptation focuses on HW dependent configuration language

- Very simple grammar:

  `<setting> = <value>;`

  `<action> = <condition statement>;

  etm.addr_comp[1].addr = 0x12340000;
  etm.addr_comp[2].accs_type = DATA_LOAD;
  ...
  etm.trace_start = etm.addr_comp[1];
  etm.trace_stop = etm.addr_comp[2] or
                  etm.addr_comp[3] or
                  etm.addr_comp[4];
  etm.trace_disable = etm.addr_rng[3] or
                     etm.addr_rng[4];
Extended Architecture Description

- Based on component/system descriptions, e.g. IP-XACT
- Contains description of
  - Debug components (ETM, ETC, ...)
  - Debug registers (memory mappings, bit fields, ...)
  - Signals / interconnects between debug components (cross triggers)
- Extensions to add semantic information
  - Debug objects:
    can be used within statements of configuration language
  - Operations:
    realize the required trace/debug functionality
  - Encodings (associated to debug objects as well as operations):
    bit field values of debug registers
Debug Objects

- Constant values (comparator values, setups, ...)
  ```c
  etm addr_comp[1].addr = 0x12340000;
  etm addr_comp[1].accs_type = DATA_LOAD;
  ```

- HW generated events (matching addresses/data/ranges, cross triggers, ...)
  ```c
  etm <action> = etm addr_comp[1];
  ```

- Virtual events which are realized using combinations of HW events (true, false, ...)
  ```c
  etm <action> = true;
  ```

- Actions (trace enable, fire cross trigger event, start counter, ...)
  ```c
  etm trace_start = <condition>;
  etm trace_dissable = <condition>;
  ```

- Debug objects may contain encodings for different registers (bit fields) to be selected depending on operations
Operations and Encodings

- Operations realize the connection between the meaning of language statements and appropriate register settings.
- Fundamental trace operations (counterparts to action objects) are composed of sub-operations.
- Operations / sub-operations defined by:
  - Their function
  - Operands (either events or other sub-operations)
- Sub-operations represent basic functionality (negation, and, or, ...) directly realized by HW.
- In the end sub-operations can be directly mapped to register settings.
- Operations / sub-operations may contain encodings for different registers (bit fields).
Operations and Encodings (cont)

- Definition of operation/sub-operation with multiple operands

- Two registers need to be set
  - Constant values
  - Values derived from operands

- Operand definitions refers to debug objects (events) and inherit encodings from them

```
<operation name="foo" type="MULTIPLE"
          operator="or">
  <registerSettings>
    <registerRef ref="register_1">
      <fieldVal ref="field_A">0x6F</fieldVal>
      <fieldVal ref="field_B">0x0</fieldVal>
    </registerRef>
    <registerRef ref="register_2">
      <fieldVal ref="addr_1">
        <operandRef ref="operand_1"/>
      </fieldVal>
      <operandRef ref="operand_1"/>
    </registerRef>
  </registerSettings>
  <operand name="operand_1">
    <eventRef ref="event_1">
      <codingRef ref="bitSelect"/>
    </eventRef>
  </operand>
  <operand name="operand_2">
    ...
  </operand>
</operation>
```
From Operations to Register Settings

- Connecting sub-operations via operands to build fundamental trace functions
- Collecting register settings by traversing the tree:
  - Register settings directly extracted
  - Register settings inherited from operands

TraceEnableEvent (0x08)

<table>
<thead>
<tr>
<th>Trace Enable Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fct</td>
</tr>
<tr>
<td>31</td>
</tr>
</tbody>
</table>

TraceEnableCtrl_1 (0x09)

<table>
<thead>
<tr>
<th>Trace Enable Ctrl_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Includes/Excludes</td>
</tr>
<tr>
<td>31 26 24 23</td>
</tr>
</tbody>
</table>

TraceStartStop (0x06)

<table>
<thead>
<tr>
<th>Stop Addr</th>
<th>Start Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Addr Comparators 1,16</td>
<td>Single Addr Comparators 1,16</td>
</tr>
<tr>
<td>31 16 15</td>
<td>0</td>
</tr>
</tbody>
</table>

ARM Developers’ Conference & Design Pavilion 2007
Conclusion
Conclusion

- Non-invasive visibility of the operation of the system required
  - Even its multi-core
  - Even on high clock rates (full speed)
- On-Chip debug and trace support (CoreSight)
  - Complex on-chip triggers and filters has to be programmed
  - Different configurations of debug system depending on SoC architecture
- High demands on tool support
  - Easy to use
  - Flexible in terms of debug target
- Adaptation of debugger using an architecture description
- Extensions to common architecture specifications (IP-XACT, ...) to bridge the gap between pure component description and functional description
Thank You

Visit us at booth 608

PLS Development Tools
50 Airport Parkway
San Jose, CA 95110-1011
USA

Phone: (408) 451 – 8408
Fax: (408) 437 – 7777
Toll Free: (877) 77DEBUG
Web: www.pls-mc.com
Email: info@pls-mc.com

ARM Developers’ Conference & Design Pavilion 2007