Platforms, ASIPs and LISATek

Federico Angiolini
federico.angiolini@unibo.it
DEIS Università di Bologna

Platform-Based Design
Competing Imperatives

- Technology push:
  - high-volume products
  - feasible design

- Marketing push:
  - fast turnaround
  - differentiated products

Two Possible Alternatives...

- Full custom design
  - Do-it-yourself: nobody knows goals better than yourself, you can do it best
  - Maximize overall design efficiency

- Platforms
  - Observation: any given space has a limited number of good solutions to challenges
  - Let’s capture them in a reusable, configurable package
What Is a Platform?

- A partial design:
  - for a particular type of system/application
  - includes embedded processor(s)
  - may include embedded software
  - customizable to customer requirements:
    - software
    - some HW components

Example: STMicro Nomadik
Platforms and Embedded Computing

- Platforms rely on embedded processors:
  - can be customized through software
  - considerable design effort can be put into CPU
- Many platforms are complex heterogeneous multiprocessors

Platform Use Methodology

- Start with reference design
- Evaluate differences required for your features
- Evaluate hardware & software changes
- Implement hardware & software changes (in parallel)
- Integrate and test
Why ASIPs

The Big Chasm: Platforms vs. Full-Custom

- Platform has few degrees of freedom:
  - harder to differentiate
  - limited flexibility
  - could simply be insufficient...
- Full-custom instead:
  - much bigger design effort
  - extremely long design cycles
  - limited reusability
- Isn’t there anything in between? :-(
Example: the Big Chasm

- We have been successfully using some pre-existing architecture (platform or full-custom)
- A brand new shiny software routine now requires adding an FPU or it will run 500% too slow
  - Platform: ouch – it is not available, and we can’t really work under the hood
  - Full-custom: ouch – it will take months to design one, then who tells our custom compiler how to use it?
- Platform is inflexible, full-custom is a mess
- What do we do??

Enter ASIPs

- Application-Specific IP cores
  - Let’s take a pretty generic IP core (for example, a general-purpose processor)
  - Let’s see what it is missing:
    - Too slow for some multimedia operation (e.g. FFT)?
    - Too limited functionality for memory management?
    - ...
  - Let’s extend it by adding instructions to its Instruction Set, and the corresponding hardware!
- Flexible (add extensions as needed by apps)
- Reusable (same basis, many flavours)
But Wait a Second... Something is Missing?

- Normally it takes a lot of effort to tweak the RTL description of a processor core!
- How do we simulate and validate the extensions? Do we also need to write a simulator?
- If I just add stuff to my core, gcc won’t know, right? So do I have to write assembly? Wait a second – the assembler doesn’t know either, yet?

Processor Description Languages and ASIP Toolchains

- Idea: let’s describe processors in a simple, ad-hoc language (i.e. neither C nor VHDL)
- This language easily allows for changing Instruction Set, register file properties, pipeline operation...
- This language also allow for automatic generation of supporting toolchains
The LISATek Flow

Available IP cores might be suboptimal for specific task at hand (e.g. multimedia acceleration)
- Customized Application-Specific IP cores (ASIPs)!

Modeling custom IP cores is time consuming
- IP core description language!

Custom IP cores require new compilation toolchain
- Automatic generation!

Silicon prototypes require effort and validation
- Automatic HDL translation!

Flow also applies to “standard” IP cores (DSP, RISC, SIMD, VLIW...)
Example: Infineon DSP Application

<table>
<thead>
<tr>
<th>LISA vs. traditional model description</th>
<th>Development Time</th>
<th>Model Lines of Code</th>
<th>Relative Area</th>
<th>Relative Clock Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional VHDL</td>
<td>5 months</td>
<td>1765</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>LISA + SystemC</td>
<td>2 months</td>
<td>738</td>
<td>+21%</td>
<td>+22%</td>
</tr>
</tbody>
</table>

Note: in fact, the datapath is automatically generated, but manual re-coding is advised

LISATek Demo

Courtesy: Sergio Foresta
Compiler Generation

Courtesy: Davide Rossi

Compiler Generation Flow
### Compiler Designer: Conventions

<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integral</td>
<td>Arg&lt;1, r2, r3, r4&gt; Rate&lt;1&gt;</td>
<td>Arg&lt;1, r2, r3, r4&gt; Rate&lt;1&gt;</td>
<td>Arg&lt;1, r2, r3, r4&gt; Rate&lt;1&gt;</td>
<td>No suitable allocatable register!</td>
</tr>
<tr>
<td>Floating</td>
<td>No such type defined</td>
<td>No such type defined</td>
<td>Arg&lt;1, r2, r3, r4&gt; Rate&lt;1&gt;</td>
<td>No suitable allocatable register!</td>
</tr>
<tr>
<td>Pointers</td>
<td>No such type defined</td>
<td>No such type defined</td>
<td>Arg&lt;1, r2, r3, r4&gt; Rate&lt;1&gt;</td>
<td>No such type defined</td>
</tr>
</tbody>
</table>

### Compiler Designer: Latency Table

#### Settings
The scheduler requires the configuration of the latency and reservation tables. Both tables can be automatically constructed during the code generation phase, or custom, usually more optimal tables can be defined below. The "Now" buttons create the tables that would be used in case of automatic generation.

- Generate latency tables automatically during code generation phase
- Generate reservation tables automatically during code generation phase

### Latency Tables | Reservation Table

<table>
<thead>
<tr>
<th>Resources</th>
<th>Data Producer</th>
<th>Data Consumer</th>
<th>Read After Write (True dependency)</th>
<th>Write After Write (Output dependency)</th>
<th>Write After Read (Anti dependency)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPC</td>
<td>F1: alu.malu</td>
<td>F1: alu.malu</td>
<td>C1: 1 2 3 4 5</td>
<td>P1: 0 0 0 0 0</td>
<td>C1: 0 0 0 0 0</td>
</tr>
<tr>
<td>C</td>
<td>F2: lb.dalu</td>
<td>F2: lb.dalu</td>
<td>C2: 1 1 1 1</td>
<td>P2: 0 0 0 0 0</td>
<td>C2: 0 0 0 0 0</td>
</tr>
<tr>
<td>FP</td>
<td>F3: stalu</td>
<td>F3: stalu</td>
<td>C3: 1 2 3 4</td>
<td>P3: 0 0 0 0 0</td>
<td>C3: 0 0 0 0 0</td>
</tr>
<tr>
<td>N</td>
<td>F4: addalu</td>
<td>F4: addalu</td>
<td>C4: 1 1 1 1</td>
<td>P4: 0 0 0 0 0</td>
<td>C4: 0 0 0 0 0</td>
</tr>
<tr>
<td>R0</td>
<td>F5: jmpalu</td>
<td>F5: jmpalu</td>
<td>C5: 1 1 1 1</td>
<td>P5: 0 0 0 0 0</td>
<td>C5: 0 0 0 0 0</td>
</tr>
<tr>
<td>R1</td>
<td>F6: bnealu</td>
<td>F6: bnealu</td>
<td>C6: 1 1 1 1</td>
<td>P6: 0 0 0 0 0</td>
<td>C6: 0 0 0 0 0</td>
</tr>
<tr>
<td>R2</td>
<td>F7: addalu</td>
<td>F7: addalu</td>
<td>C7: 1 1 1 1</td>
<td>P7: 0 0 0 0 0</td>
<td>C7: 0 0 0 0 0</td>
</tr>
<tr>
<td>R3</td>
<td>F8: stalu</td>
<td>F8: stalu</td>
<td>C8: 1 1 1 1</td>
<td>P8: 0 0 0 0 0</td>
<td>C8: 0 0 0 0 0</td>
</tr>
<tr>
<td>R4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Compiler Designer: Reservation Table

- Settings
  The scheduler requires the configuration of the latency and reservation tables. Both tables can be automatically constructed during the code generation phase, or, custom, usually more optimal tables can be defined below. The "Now" buttons create the tables that would be used in case of automatic generation.

  - Generate latency tables automatically during code generation phase
  - Generate reservation tables automatically during code generation phase

- Latency Tables
- Reservation Tables

(Virtual) Resources | Resource Templates
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>INSTRUCTION == alu Ri</td>
</tr>
<tr>
<td>T2</td>
<td>INSTRUCTION == alu Ri</td>
</tr>
<tr>
<td>T3</td>
<td>INSTRUCTION == alu Ri</td>
</tr>
<tr>
<td>T4</td>
<td>INSTRUCTION == alu Rr</td>
</tr>
<tr>
<td>T5</td>
<td>INSTRUCTION == branch</td>
</tr>
<tr>
<td>T6</td>
<td>INSTRUCTION == branch</td>
</tr>
<tr>
<td>T7</td>
<td>INSTRUCTION == ciao Ri</td>
</tr>
<tr>
<td>T8</td>
<td>INSTRUCTION == ciao Rr</td>
</tr>
<tr>
<td>T9</td>
<td>INSTRUCTION == li Rr</td>
</tr>
<tr>
<td>T10</td>
<td>INSTRUCTION == li Rr</td>
</tr>
<tr>
<td>T11</td>
<td>INSTRUCTION == li Rr</td>
</tr>
<tr>
<td>T12</td>
<td>INSTRUCTION == nop</td>
</tr>
<tr>
<td>T13</td>
<td>INSTRUCTION == str Ri</td>
</tr>
<tr>
<td>T14</td>
<td>INSTRUCTION == trap</td>
</tr>
</tbody>
</table>

Compiler Designer: Matcher (1)
Compiler Designer: Matcher (2)

This rule performs addition on two general purpose registers and writes the result to a register.

Example C-code that would match this pattern:
```
sll r3, r1, 1;
```

Rule Documentation

This rule performs addition on two general purpose registers and writes the result to a register.

Example C-code that would match this pattern:
```
sll r3, r1, 1;
```