OpenSource RISC Processors: An introduction into OpenRISC and RISC-V

Advanced System-on-Chip Design
Lecture 6
05.04.2016

Michael Gautschi
IIS-ETHZ

Prof. Luca Benini
IIS-ETHZ
Introduction – the “best” core

- Single issue in-order is most energy efficient
- Put more than one + shared memory to fill cluster area
Introduction – Building PULP

SIMD + MIMD + sequential

Private per-core instruction cache

4-stage, in-order OpenRISC / RISC-V core

DEMUX

LOW LATENCY INTERCONNECT

1 Cycle Shared Multi-Banked L1 Data Memory + Low Latency Interconnect

“GPU like” shared memory → low overhead data sharing

Near Threshold but parallel → Maximum Energy efficiency when Active

+ strong power management for (partial) idleness

2 ..16 Cores

Double buffering

Tightly Coupled DMA

DMA

LOW LATENCY INTERCONNECT

L1 TCDM

Periph
+ExtM

PE0

I$0

.....

I$N-1

PEN-1

MB0

MBM-1

05.04.2016
Introduction – PULP Architecture

- Uses Open Source RISC processor
  - OpenRISC, RISC-V ISA
Introduction

• Outline:
  – OpenRISC Instruction-set
    • Basic instruction set
  – Micro-architecture
    • Organization of the pipeline
  – Instruction set extensions for improved performance
    • Hardware and software impact
  – RISC-V architecture
    • Difference to OpenRISC
  – Exercise session about OpenRISC/ RISC-V processor cores
    • Exercise session

• Goals:
  – Learn how to run applications on the Pulpino architecture
  – Understand the impact of the presented hardware extensions
    • Including some pro/ and cons
OpenRISC Instruction Set

- Open source 32-/64bit RISC architecture
  - Similar to MIPS architecture described in Hennessey/Patterson

- ORBIS32:
  - 32-bit integer instruction
  - 32-bit load/store instructions
  - Program flow instructions

- ORBIS64:
  - 64-bit integer instructions
  - 64-bit load/store instructions

- ORFPX32:
  - Single precision floating point instructions

- ORFPX64:
  - Double-precision floating point instructions

- ORVDX64:
  - 64-bit vector instructions

⇒ In the following we focus on the 32-bit ORBIS32 instruction set!
OpenRISC Instruction Set

- ORBISX32 consists of three types of instructions
  - R-type instructions:
    - Register - register operations
    - Examples:
      - ALU operations: \(l\text{.add}, l\text{.mul}, l\text{.sub}, l\text{.or}, l\text{.mac},\) etc.
      - Comparisons: \(l\text{.sfeq}, l\text{.sfges},\) etc.

---

### l.add

**Add**

<table>
<thead>
<tr>
<th>31</th>
<th>26 25</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>9</th>
<th>8 7</th>
<th>4 3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode 0x38</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>reserved</td>
<td>opcode 0x0</td>
<td>reserved</td>
<td>opcode 0x0</td>
<td></td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>1 bit</td>
<td>2 bits</td>
<td>4 bits</td>
<td>4 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### l.sfeq

**Set Flag if Equal**

<table>
<thead>
<tr>
<th>31</th>
<th>21 20</th>
<th>16 15</th>
<th>11 10</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode 0x720</td>
<td>A</td>
<td>B</td>
<td>reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>11 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
OpenRISC Instruction Set

• ORBISX32 consists of three types of instructions
  
  – I-type instructions:
    • Operations with an immediate
    • Examples:
      – Load/store operations: \( l.lwz, l.sw, l.lhz, l.lbz \) etc.
      – ALU operations: \( l.addi, l.muli, l.ori \) etc.
      – Comparisons: \( l.sfeqi, l.sfnei \) etc.

\( l.lwz \)  Load Single Word and Extend with Zero  \( l.lwz \)

\[
\begin{array}{cccccccccc}
\hline
\text{opcode} & 0x21 & D & A & I \\
\text{bits} & 6 & 5 & 5 & 16 \\
\end{array}
\]

\( l.muli \)  Multiply Immediate Signed  \( l.muli \)

\[
\begin{array}{cccccccccc}
\hline
\text{opcode} & 0x2c & D & A & I \\
\text{bits} & 6 & 5 & 5 & 16 \\
\end{array}
\]
OpenRISC Instruction Set

- ORBISX32 consists of three types of instructions
  - J-type instructions:
    - Jumps and branches
    - Examples:
      - Jump instructions: \(l.j\), \(l.jal\), \(l.rfe\), etc.
      - Conditional branches: \(l.bf\), \(l.bnf\)

\[
\begin{align*}
\textbf{l.jal} & \quad \text{Jump and Link} & \quad \textbf{l.jal} \\
\begin{array}{|c|c|c|}
\hline
31 & \ldots & 26 \ 25 \\
\hline
\text{opcode} & 0\times1 & N \\
\hline
\text{6 bits} & \text{26 bits} & \\
\hline
\end{array}
\end{align*}
\]

\[
\begin{align*}
\textbf{l.bf} & \quad \text{Branch if Flag} & \quad \textbf{l.bf} \\
\begin{array}{|c|c|c|}
\hline
31 & \ldots & 26 \ 25 \\
\hline
\text{opcode} & 0\times4 & N \\
\hline
\text{6 bits} & \text{26 bits} & \\
\hline
\end{array}
\end{align*}
\]
OpenRISC Micro-architecture:

- Core architecture which has been originally developed here at IIS in a semester thesis.
  - The architecture is called OR10N
  - It has been improved over the years and become a good core architecture

- Simple four-stage pipeline architecture: IF, ID, EX, WB
- Single cycle memory access
Register file & special purpose registers (SPR)

• Register file organization:
  – 32 registers 32-bit registers
  – Most important registers are:
    • r0 = always zero
    • r1 = stack pointer
    • r9 = link register, holds function return address
    • r11/r12 = return values

• Special purpose registers:
  – Status register contains flags \{overflow, carry, branch\}
  – Contains registers which are not regularly accessed:
    • Interrupt controller configuration
    • Timer
    • Data/instruction cache control
  – Debug unit
  – Performance counters
Load/Store Unit

- 32 bit load-store interface

Supported instructions:
  - Load word/halfword/byte
    - With zero or sign extension
  - Addressing mode aligned data requests
    - l.lwz/s word aligned
    - l.lhz/s half word aligned
    - l.lbz/s byte aligned
  - Stall pipeline if exception has been detected
    - Access to protected address
    - Unaligned access
  - No out of order requests
OpenRISC: Control Flow

- **Branches**
  - `l.bnf`: jump to PC + sign extended immediate if flag is not set
  - `l.bf`: jump to PC + sign extended immediate if flag is set
  - Delay slot is always executed

- **Jumps**
  - `l.jr`: jump to address stored in a register
  - `l.jalr`: jump to address stored in a register and link r9 to instruction after delay slot
  - `l.j`: jump to PC + sign extended immediate
  - `l.jal`: jump to PC + sign extended immediate and link r9
  - `l.rfe`: return from exception, jump to EPCR

- **No support for VLIW**
  - Instructions are always 32 bit
OR10N Instruction Extensions for OR10N:

- In order to improve performance and efficiency of the core we have evaluated several instructions and added the following instructions:
  - Hardware loops
  - Pre/post memory address update
  - New MAC
  - Vector unit
  - Unaligned memory access
Instruction Extensions: Hardware Loops

- Hardware loops or Zero Overhead Loops can be implemented to remove the branch overhead in for loops.
- After configuration with start, end, count variables no more comparison and branches are required.
- Smaller loop benefit more!

- Loop needs to be set up beforehand and is fully defined by:
  - Start address
  - End address
  - Counter

9 loop instructions
3 setup instructions +
7 loop instructions
Instruction Extensions: Hardware Loops

- Two sets registers implemented to support nested loops.

- Area costs:
  - Processor core area increases by 5%

- Performance:
  - Speedup can be up to factor 2!

  - Hardware loop setup with:
    - 3 separate instructions
      \( lp\text{-}\text{start}, \ lp\text{-}\text{end}, \ lp\text{-}\text{count}, \ lp\text{-}\text{counti} \)
      \( \Rightarrow \) No restriction on start/end address

    - Fast setup instructions
      \( lp\text{-}\text{setup}, \ lp\text{-}\text{setupi} \)
      \( \Rightarrow \) Start address = PC + 4
      \( \Rightarrow \) End address = start address + offset
      \( \Rightarrow \) Counter from immediate/register

<table>
<thead>
<tr>
<th>Instruction format and Opcode</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( lp\text{-}\text{start} ), ( S ) ( \text{eg.} \ \text{lp\text{-}start} \ L0, 10 )</td>
<td>( \text{HWLP_START}{J} = \text{sex}{S} + \text{PC} )</td>
</tr>
<tr>
<td>( S )</td>
<td>( \text{HWLP_END}{J} = \text{sex}(E) + \text{PC} )</td>
</tr>
<tr>
<td>( \text{lp\text{-}counti} ), ( C ) ( \text{eg.} \ \text{lp\text{-}counti} \ L0, 8 )</td>
<td>( \text{HWLP_COUNT}{J} = \text{sex}(\text{C}) )</td>
</tr>
<tr>
<td>( C )</td>
<td>( \text{HWLP_COUNT}{J} = [\text{rA}] )</td>
</tr>
<tr>
<td>( \text{lp\text{-}setup} ), ( \text{E} ), ( \text{C} ) ( \text{eg.} \ \text{lp\text{-}setup} \ L0, 4, 8 )</td>
<td>( \text{HWLP_START}{J} = \text{PC} + 4 )</td>
</tr>
<tr>
<td>( \text{HWLP_END}{J} = \text{sex}(E) + \text{PC} )</td>
<td></td>
</tr>
<tr>
<td>( \text{HWLP_COUNT}{J} = \text{sex}(C) )</td>
<td></td>
</tr>
<tr>
<td>( \text{lp\text{-}setup} ), ( \text{E} ), ( \text{rA} ) ( \text{eg.} \ \text{lp\text{-}setup} \ L0, 8, 5 )</td>
<td>( \text{HWLP_START}{J} = \text{PC} + 4 )</td>
</tr>
<tr>
<td>( \text{HWLP_END}{J} = \text{sex}(E) + \text{PC} )</td>
<td></td>
</tr>
<tr>
<td>( \text{HWLP_COUNT}{J} = [\text{rA}] )</td>
<td></td>
</tr>
</tbody>
</table>
Instruction Extensions: Post increment

- **Automatic address update**
  - Update base register with computed address after the memory access
  ⇒ Save instructions to update address register
  - Post-increment:
    • Base address serves as memory address

- **Offset can be stored in:**
  - Register
  - Immediate

⇒ save 2 additional instructions to track the read address of the operands!
Instruction Extensions: Post increment

- Register file requires additional write port
- Register file requires additional read port if offset is stored in register
- Processor core area increases by 8-12 %
  - Ports can be used for other instructions
New MAC: Accumulation on register file

- **Old MAC:**
  - Accumulation on special 64 bit register

- **New MAC:**
  - Accumulation only on 32 bit data
  - Directly on the register file

- **Pro:**
  - Faster access to mac accumulation
  - Many accumulations in parallel
  - Single cycle mult/mac

- **Contra:**
  - Additional read port on the register file
    - can be used for pre/post increment with register
Instruction Extensions: 32 bit Vector Support

- Vector modes: (bytes, halfwords, word)
  - 4 byte operations
    - With byte select
  - 2 halfword operations
    - With halfword select
  - 1 word operation

- Vector ALU supports:
  - Vector additions
  - Vector subtractions
  - Vector comparisons:
    - Rise flag if *any*, or *all* conditions are true

- Fused vector Mult/Mac:
  - Dynamic range problem in vector multiplications

Vectorial Adder:
32 bit Vector Support: Vectorial Multiplication

- Multiplication on
  - Word,
  - Halfword
  - Byte

- 3 different multipliers:
  - $4^*$ (8x8=8) mults.
  - $2^*$ (16x16=16) mults.
  - $1^*$ (32x32=32) mult.

- Output has same dynamic range as input!
  - Overflows fast in 8bit case
Performance Improvements:

Hardware loops (H) & pre-/post increment (IH):
Up to 1.75x speedup

Optimized MAC (IHM)
Up to 2.25x speedup

Vector extensions (IHVM)
Up to 5x speedup but only rarely used => Dot product

Cortex M4: STM32F429ZI
Slower, no intrinsic used
32 bit Vector Support: Dot Product Multiplier

- Dot Product: (half word example)

=> 2 multiplications, 1 addition, 1 accumulation in 1 cycle!
Instruction Extension: Fractional Support

- Fractional format in Q-format [1]
  - Q3.12 means 1 sign bit, 3 integer bits and + 12 fractional bits (16 bit in total)
  - Q3.12 format in a 32 bit reg:

- Multiplication in Q format requires a shifter:
  - Q3.12 * Q3.12 =
  - Q3.12 * Q3.12 >> 12 =

---

32 bit Vector Support: Full Multiplier Architecture

16b Dot Product Mult.
Fractional Multiplier

8b Dot Product Mult.

Integer Multiplier
32 bit Vector Support: Full Multiplier Architecture

• Supports:
  – Fractional multiplications
  – Dot Products for 16b vectors
  – Dot Products for 16b vectors
  – Integer multiplication 32b*32b into 32b

• Multiplier costs:
  – Up to 2.3x bigger without resource sharing
  – Sharing makes arch. slower
  – Power ?
32 bit Vector Support: Power Reduction

- **Operand Isolation**
  - Gating inputs to zero

- **Separate input registers:**
  - Eliminating all active power of unused units
  - 30-60% power reduction with respect to a design without separate input registers
32 bit Vector Support: Unaligned memory access

- Unaligned memory access with 32 bit data interface:
  - Difficult to read/write unaligned words, because memories are 32 bit wide
  - Possible with multibanked memories
    - But significant hardware costs
    - Area and timing

- Implemented with two subsequent memory requests

Example: stencil with vector
32 bit Vector Support: Shuffle Instruction

- In order to use the vector unit the elements have to be aligned in the register file
- Shuffle allows to recombine bytes into 1 register:
  - `lv.shuffle2.b rD, rA, rB`
    - \( rD\{3\} = (rB[26]==0) \ ? \ rA:rD \) \( rB[25:24]\)
    - \( rD\{2\} = (rB[18]==0) \ ? \ rA:rD \) \( rB[17:16]\)
    - \( rD\{1\} = (rB[10]==0) \ ? \ rA:rD \) \( rB[9:8]\)
    - \( rD\{0\} = (rB[ 2]==0) \ ? \ rA:rD \) \( rB[1:0]\)
    - With \( rX\{i\} = rX[(i+1)*8-1:i*8]\)
32 bit Vector Support: Summary

• ALU extensions for packed SIMD are easy
  • Vector addition, comparisons, etc.

• Multiplier is more complicated. Dot product is very useful for simple kernels
  • convolutions, multiplications etc.

• Load store unit needs support for unaligned access
  • Good trade-off with support in two cycles

• Vector recombination instructions are very important for the compiler!
  • To recombine bytes/ halfwords directly in registers instead of reloading everything from memory
The RISC-V ISA

- Modern ISA created by UC Berkeley for their research
- Available for 32-bit, 64-bit and 128-bit
- Little-endian
- Published as Free and Open RISC ISA

- The ISA specifications were previously controlled by UCB, now shifting to the RISC-V foundation

- RISC-V foundation is controlled by the members
  Everyone can become member, just costs a bit of money

- See official website http://riscv.org
The RISC-V ISA: Introduction

• Generally kept very simple and extendable

• Separated into multiple specifications
  – User-Level ISA spec (compute instructions)
  – Compressed ISA spec (16-bit instructions)
  – Privileged ISA spec (supervisor-mode instructions)
  – More to come

• Implementations:
  – We have a similar architecture than in OpenRISC
  – Can be used in Semester/Master - Projects
User-Level ISA Spec

- Defines the normal instructions needed for computation
- Spec separated into “extensions”
- Defines a mandatory base integer instruction set: “I extension”
- ISA support is given by RV + word-width + extensions supported
  - E.g. **RV32I** means 32-bit RISC-V with support for the I instruction set

- **I**: Integer instructions; alu, branches, jumps, loads and stores
  - Support for misaligned memory access is mandatory
- **M**: Multiplication and (!!!) Division
- **A**: Atomic instructions
- **F**: Single-Precision Floating-Point
- **D**: Double-Precision Floating-Point
- **C**: Compressed Instructions (more later)
User-Level ISA Spec: Standard and Non-standard extensions

- Extensions mentioned so far are so called **Standard Extensions**
- Reserved opcodes for standard extensions
- Rest of opcodes free for non-standard implementations
- Standard extensions will be frozen and will not change in the future
Instruction Length Encoding

- Supports by design 16, 32, 48, 64, ... bit long instruction words

\[
\begin{align*}
\text{xxxxxxx} & \text{xxxxxx}a \quad 16\text{-bit (aa} \neq 11) \\
\text{xxxxxxx} & \text{xxxxx}bbb11 \quad 32\text{-bit (bbb} \neq 111) \\
\cdots \text{xxxx} & \text{xxxxxx}011111 \quad 48\text{-bit} \\
\cdots \text{xxxx} & \text{xxxxxx}011111 \quad 64\text{-bit} \\
\cdots \text{xxxx} & \text{xxxxnnn}111111 \quad (80+16\times\text{nnn})\text{-bit, nnn} \neq 1111 \\
\cdots \text{xxxx} & \text{xxxx1111111111} \quad \text{Reserved for} \geq 320\text{-bits}
\end{align*}
\]

Byte Address: base+4 \hspace{2cm} base+2 \hspace{2cm} base
Compressed Instruction Spec (Draft)

• Still a draft, but will be frozen very soon if there are no complaints

• Compressed instructions are 16-bit wide
  – Allows to reduce code size
  – needs support for misaligned instruction memory access

• Compressed instruction ISA spec is no ISA per se
  – All compressed instructions map to I instructions, can be expanded
  – Preprocessing step for instructions needed or separate decoding for compressed instructions
Compressed Instruction Spec (Draft)
Claim code size reduction by ~34%
Differences to OpenRISC: No flags, no delay slot

• Branches:

<table>
<thead>
<tr>
<th>OpenRISC</th>
<th>RISC-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>l.sfeq rA, rB</td>
<td>beq rA, rB, 0x20</td>
</tr>
<tr>
<td>l.bf 0x20</td>
<td></td>
</tr>
<tr>
<td>l.nop</td>
<td></td>
</tr>
</tbody>
</table>

• Jumps are more general

<table>
<thead>
<tr>
<th>OpenRISC</th>
<th>RISC-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>l.jalr rB</td>
<td>jalr rD, rs1, 0x10</td>
</tr>
</tbody>
</table>

05.04.2016
Differences to OpenRISC: Opcode Sizes

- **RISC-V**: 7-bit main opcodes
  - actually only 5-bits due to compressed instructions
  - sub-opcodes (funct*) for most instruction types

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R-type</td>
</tr>
<tr>
<td>rs2</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>rs1</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>funct3</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>rd</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>opcode</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>

<table>
<thead>
<tr>
<th>imm[11:0]</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>I-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>imm[31:12]</th>
<th>rd</th>
<th>opcode</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>U-type</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Differences to OpenRISC: Immediate Size

- **RISC-V**: 12-bit immediates, all sign-extended
  - sign-bit always in same position

- **OpenRISC**: 16-bit immediates, mixed sign- and zero-extended

<table>
<thead>
<tr>
<th>Function</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
<th>Immediate Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>31</td>
<td>25</td>
<td>24</td>
<td>20</td>
<td>19</td>
<td>imm[11:0]</td>
</tr>
<tr>
<td>I-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>imm[11:5]</td>
</tr>
<tr>
<td>U-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>imm[31:12]</td>
</tr>
</tbody>
</table>
Differences to OpenRISC: Constructing a 32-bit number

- OpenRISC has 16 bit immediates with zero-extension
  - Just use 2 instructions

- RISC-V has 12 bit immediates with sign-extension
  - load upper immediate first and then add sign-extended value
  - upper immediate is set to imm+1 to correct for sign-extension if necessary

<table>
<thead>
<tr>
<th>OpenRISC</th>
<th>RISC-V</th>
</tr>
</thead>
<tbody>
<tr>
<td>l.movhi r1, 0x1000</td>
<td>lui x1, 0x10002</td>
</tr>
<tr>
<td>l_ori r1, r1, 0x1800</td>
<td>add x1, x1, -2048</td>
</tr>
</tbody>
</table>
RI5CY: Core architecture (4 stage pipeline)

- CSR in EX instead of WB
RI5CY: Prefetcher: Why is branching difficult

• No delay slot in RISC-V:
  – Jumps loose one cycle
    • Next instruction already fetching and probably ready in IF stage

• Combined branches: No setflag instruction
  – Branch decision computed with branching instruction
  – Branch decision computed in EX stage
    – Taken branches loose two cycles
      • Branch only available late in pipeline
    – Not taken branches don’t lose cycles
RI5CY: Prefetcher: The need for prefetching

- Instruction memory is word-aligned
  - Does not accept misaligned accesses

- Cross-word instruction needs to be assembled from two words

- If lower half word is compressed, no need to fetch next word already

- Solution: Prefetcher with storage for >2 words
  - We choose 4 words (1 cacheline) for optimal performance and to deal with cache misses
RI5CY: Fully independent pipeline

- Ready & valid signals running left and right respectively
- Pipeline stages can be empty
  - Not possible in OR10N
- Easy to integrate multi-cycle instructions in each stage
  - Similar for “limited” out-of-order execution
RI5CY: Simulation Checker

- Instruction Set Simulator running in parallel using instruction and data access inputs from RTL
- After every cycle write-back data to RF is compared
- ISS serves as golden model
- Could also execute a random instruction stream, since check is automatically and implicitly done

- The simchecker is not activated by default
- To activate:
  - Uncomment `define simchecker` in riscv_defines.sv
  - Uncomment sv_lib loading in vsim setup files
  - Build and copy rtl_checker from SDK to vsim work directory
RI5CY: Random Stall Injection

- Only on PULPino for the moment
- Random latency (non-synthesizable) introduced to instruction and data accesses
- Checks core interfaces in a way that we would seldom see on the platform

- To activate
  - Uncomment `define DATA_STALL_RANDOM and `define INSTR_STALL_RANDOM in config.sv of PULPino
RI5CY: Tracer

- Similar to objdump output
- Displays also read and written registers
  - x??: read value
  - x??= written value
  - PA: physical address for memory accesses
- Virtual platform follows same output format
- Files automatically generated for RTL simulation
  - trace_core_xx_xx.log

<table>
<thead>
<tr>
<th>Timestamp</th>
<th>Cyc.</th>
<th>PC</th>
<th>Instr.</th>
<th>Assembler</th>
<th>String</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>24530000</td>
<td>ps</td>
<td>1197</td>
<td>0000002c</td>
<td>faddce3</td>
<td>bge</td>
<td>x27, x26, -8</td>
</tr>
<tr>
<td>24510000</td>
<td>ps</td>
<td>1196</td>
<td>00000128</td>
<td>004d0d13</td>
<td>addi</td>
<td>x26, x26, 4</td>
</tr>
<tr>
<td>24590000</td>
<td>ps</td>
<td>1200</td>
<td>00000124</td>
<td>000d2023</td>
<td>sw</td>
<td>x0, 0(x26)</td>
</tr>
<tr>
<td>24630000</td>
<td>ps</td>
<td>1202</td>
<td>0000012c</td>
<td>faddce3</td>
<td>bge</td>
<td>x27, x26, -8</td>
</tr>
<tr>
<td>24610000</td>
<td>ps</td>
<td>1201</td>
<td>00000128</td>
<td>004d0d13</td>
<td>addi</td>
<td>x26, x26, 4</td>
</tr>
<tr>
<td>24650000</td>
<td>ps</td>
<td>1203</td>
<td>00000130</td>
<td>00000513</td>
<td>addi</td>
<td>x10, x0, 0</td>
</tr>
<tr>
<td>24670000</td>
<td>ps</td>
<td>1204</td>
<td>00000134</td>
<td>00100593</td>
<td>addi</td>
<td>x11, x0, 1</td>
</tr>
</tbody>
</table>
RI5CY: Performance Counters

- Events to be counted:
  - #cycles
  - #instructions
  - #ld_stall: load data hazards
  - #jr_stall: number of jump register data hazards
  - #imiss: cycles waiting for instructions
  - #ld
  - #st
  - #jump
  - #branch: total number of branches (w/o jumps)
  - #btaken: branches that were taken
  - #rvc: number of rvc insns
  - #ld_ext: LD to non-tcdm
    - misaligned access counted twice
  - #st_ext: ST to non-tcdm
  - #ld_ext_cyc
  - #st_ext_cyc
  - #tcdm_cont: cycles wasted due to waiting for grants in L1
RI5CY: Performance Counters

- On ASIC only one counter + one register
- On FPGA/RTL sim
  - one counter + one register per metric
- Binary tracer for performance counters is not yet available (for KCG)
Other RISC-V Cores

Z-Scale / V-Scale

- From UC Berkeley
- Their take on a small core
- Z-Scale: Written in Chisel
- V-Scale: Written in Verilog
- Z-Scale & V-Scale virtually identical

Mini project:
Compare V-Scale/Z-Scale to our core [1]

Semester project:
Design a mini core with lower power consumption than V-Scale, Z-Scale

Z-scale Pipeline

<table>
<thead>
<tr>
<th>Category</th>
<th>ARM Cortex-M0</th>
<th>RISC-V Zscale</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA</td>
<td>32-bit ARM v6</td>
<td>32-bit RISC-V (RV32IM)</td>
</tr>
<tr>
<td>Architecture</td>
<td>Single-Issue In-Order 3-stage</td>
<td>Single-Issue In-Order 3-stage</td>
</tr>
<tr>
<td>Performance</td>
<td>0.87 DMIPS/MHz</td>
<td>1.35 DMIPS/MHz</td>
</tr>
<tr>
<td>Process</td>
<td>TSMC 40LP</td>
<td>TSMC 40GPLUS</td>
</tr>
<tr>
<td>Area w/o Caches</td>
<td>0.0070 mm²</td>
<td>0.0098 mm²</td>
</tr>
<tr>
<td>Area Efficiency</td>
<td>124 DMIPS/MHz/mm²</td>
<td>138 DMIPS/MHz/mm²</td>
</tr>
<tr>
<td>Frequency</td>
<td>≤50 MHz</td>
<td>~500 MHz</td>
</tr>
<tr>
<td>Voltage (RTV)</td>
<td>1.1 V</td>
<td>0.99 V</td>
</tr>
<tr>
<td>Dynamic Power</td>
<td>5.1 µW/MHz</td>
<td>1.8 µW/MHz</td>
</tr>
</tbody>
</table>
Other RISC-V Cores: Rocket

- From UC Berkeley, written in Chisel
- 64-Bit Implementation

<table>
<thead>
<tr>
<th>Category</th>
<th>ARM Cortex-A5</th>
<th>RISC-V Rocket</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA</td>
<td>32-bit ARM v7</td>
<td>64-bit RISC-V v2</td>
</tr>
<tr>
<td>Architecture</td>
<td>Single-Issue In-Order</td>
<td>Single-Issue In-Order 5-stage</td>
</tr>
<tr>
<td>Performance</td>
<td>1.57 DMIPS/MHz</td>
<td>1.72 DMIPS/MHz</td>
</tr>
<tr>
<td>Process</td>
<td>TSMC 40GPLUS</td>
<td>TSMC 40GPLUS</td>
</tr>
<tr>
<td>Area w/o Caches</td>
<td>0.27 mm²</td>
<td>0.14 mm²</td>
</tr>
<tr>
<td>Area with 16K Caches</td>
<td>0.53 mm²</td>
<td>0.39 mm²</td>
</tr>
<tr>
<td>Area Efficiency</td>
<td>2.96 DMIPS/MHz/mm²</td>
<td>4.41 DMIPS/MHz/mm²</td>
</tr>
<tr>
<td>Frequency</td>
<td>&gt;1GHz</td>
<td>&gt;1GHz</td>
</tr>
<tr>
<td>Dynamic Power</td>
<td>&lt;0.08 mW/MHz</td>
<td>0.034 mW/MHz</td>
</tr>
</tbody>
</table>
Other RISC-V Cores

BOOM: Berkeley Out-of-Order Processor

- From UC Berkeley, written in Chisel
- Parametrizable for Dual-Issue/Quad-Issue

<table>
<thead>
<tr>
<th>Category</th>
<th>ARM Cortex-A9</th>
<th>RISC-V BOOM-2w</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA</td>
<td>32-bit ARM v7</td>
<td>64-bit RISC-V v2 (RV64G)</td>
</tr>
<tr>
<td>Architecture</td>
<td>2 wide, 3+1 issue Out-of-Order 8-stage</td>
<td>2 wide, 3 issue Out-of-Order 6-stage</td>
</tr>
<tr>
<td>Performance</td>
<td>3.59 CoreMarks/MHz</td>
<td>3.91 CoreMarks/MHz</td>
</tr>
<tr>
<td>Process</td>
<td>TSMC 40GPLUS</td>
<td>TSMC 40GPLUS</td>
</tr>
<tr>
<td>Area with 32K caches</td>
<td>~2.5 mm²</td>
<td>~1.00 mm²</td>
</tr>
<tr>
<td>Area efficiency</td>
<td>1.4 CoreMarks/MHz/mm²</td>
<td>3.9 CoreMarks/MHz/mm²</td>
</tr>
<tr>
<td>Frequency</td>
<td>1.4 GHz</td>
<td>1.5 GHz</td>
</tr>
<tr>
<td>Power</td>
<td>0.5-1.9 W (2 cores + L2) @ TSMC 40nm, 0.8-2.0 GHz</td>
<td>0.25 W (1 core + L1) @ TSMC 45nm, 1 GHz</td>
</tr>
</tbody>
</table>

Master project: [1]
Design and implement a VLIW-architecture supporting the RISC-V ISA

Mini Project:
Initial design consideration for the implementation (pro/cons evaluation)
Exercise session:

• In the exercise we are going to cover:

  – How to compile and run an application using:
    • The open source Pulpino platform

  – Impact of the new instructions:
    • Hardware loops
    • Pre/post increment
    • Vector support
    • Dot product
    • Shuffle instruction

  – How to use highly optimized kernels in programs
    • Convolutions

  – Comparison of RISC-V to ARM Cortex M4
Q&A