OpenRISC Processor: An Introduction to the Basic and Extended Instruction-Set

Advanced System-on-Chip Design

23.03.2015

Michael Gautschi
IIS-ETHZ

Prof. Luca Benini
IIS-ETHZ
Introduction

• Contents:
  – OpenRISC Instruction-set
    • Basic instruction set
  – Micro-architecture
    • Organization of the pipeline
    • Interrupt, event and debug support in hardware
  – Instruction set extensions for improved performance
    • Hardware and software impact
  – Exercise session about OpenRISC processor core
    • Exercise session

• Goals:
  – Knowing the basic instructions of the OpenRISC architecture
  – Learn how to use the compiler, simulator and RTL-simulations
  – Understand the impact of the presented simple 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 instruction

- 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: \( l.add, l.mul, l.sub, l.or, l.mac, \text{etc.} \)
      - Comparisons: \( l.sfeq, l.sfges, \text{etc.} \)
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  \( l.lwz \)

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opcode 0x21</th>
<th>D</th>
<th>A</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opcode 0x2c</th>
<th>D</th>
<th>A</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
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: \( l.bf, l.bnf \)

\[
\begin{array}{|c|c|c|}
\hline
\text{l.jal} & \text{Jump and Link} & \text{l.jal} \\
\hline
31 & \ldots & 26 \\
\hline
\text{opcode 0x1} & \text{N} \\
\hline
6 \text{ bits} & 26 \text{ bits} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
\text{l.bf} & \text{Branch if Flag} & \text{l.bf} \\
\hline
31 & \ldots & 26 \\
\hline
\text{opcode 0x4} & \text{N} \\
\hline
6 \text{ bits} & 26 \text{ bits} \\
\hline
\end{array}
\]
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
    • Mac unit
    • Data/instruction cache control
  – Debug unit
  – Performance counters
Load/Store Unit

- 32 bit load-store interface of processor

- Supported instructions:
  - Load word/half word/ 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.start`, `lp.end`, `lp.count`, `lp.counti`
      ⇒ No restriction on start/end address
  - Fast setup instructions
    - `lp.setup`, `lp.setupi`
      ⇒ Start address = PC + 4
      ⇒ End address = start address + offset
      ⇒ Counter from immediate/register

### Instruction format and Opcode

<table>
<thead>
<tr>
<th>Instruction format and Opcode</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lp.start J, S</code> (eg. <code>lp.start L0, 10</code>)</td>
<td><code>HWLP_START[J]= sext(S+4)+PC</code></td>
</tr>
<tr>
<td><code>000010 000 JJ SSSSSSSSSSSSSSSS</code></td>
<td></td>
</tr>
<tr>
<td><code>lp.end J, S</code> (eg. <code>lp.end L0, -8</code>)</td>
<td><code>HWLP_END[J] = sext(E+4)+PC</code></td>
</tr>
</tbody>
</table>
| `000010 001 JJ EEEEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEBEB
Instruction Extensions: Pre/post increment

- Automatic address update
  - Update base register with computed address after the memory access
  - Save instructions to update address register

- Pre-increment:
  - Base address + offset serves as memory address

- Post-increment:
  - Base address serves as memory address

- Offset can be stored in:
  - Register
  - Immediate

⇒ save 2 additional instructions to track the address of the operands to read!
Instruction Extensions: Pre/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
Performance Improvements:

- Pre/post increment improves performance in almost all applications
- First hardware loop brings the largest boost
New MAC: Accumulation on register file

• 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: 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 supports:
  – Vector multiplications
  – Vector multiply-accumulate (mac)
  – Results have the same dynamic range as inputs
    • 64bit multiplication result can be obtained via software
Vector support example

• Example:
  – Assume perfect vectorizable code:

    ```
    char result[N];
    char A[N];
    char B[N];
    
    for (i = 0; i < N; i++) {
        result[i] = A[i] + B[i];
    }
    ```

  • What speedup do you expect?
    – 4 times faster addition
    – 4 times faster memory access

Without vector:
  1 setupi + 2N lbz + N sb + N add
With vector:
  1 setupi + N/2 lwz + N/4 sw + N/4 add

=> Factor 4 speedup!

• What if:

    ```
    char result[N+4];
    char A[N+4];
    char B[N+4];
    
    for (i = 1; i < N+1; i++) {
        result[i] = A[i] + B[i];
    }
    ```

  • Data is not aligned anymore!
  • What speedup do you expect now?

Without vector:
  1 setupi + 2N lbz + N sb + N add
With vector:
  1 setupi + ? lbz/lwz + ? Sb + N/4 add

=> much smaller speedup!
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

1 word

1 unaligned word
Debug support for OR10N

- Features in debug mode:
  - Access to general purpose registers (GPR)
  - Access to special purpose register (SPR)
    - GPR and SPR are connected to debug signals in debug mode (muxed)
  - No watchpoints
  - Read/write program counter
  - Step through code

- Debug unit is connected to advanced debug unit of PULP
- Debug mode is activated when:
  - Trap instruction is decoded and triggers breakpoint to inform debug unit
  - Debug unit receives external stall `dbg_stall`
- Memory can be accessed through dedicated axi-port on adv. debug unit
Event and Interrupt

- Events to wake up core from ultra-low power sleep state
- Interrupts to handle irregular “exceptions”

- Tasks of the event unit:
  - Mask interrupts
  - Mask events
  - Clock gate core if it enters sleep mode
    - Only if core is in stable state
  - Send event to wake up cores
  - Masks are read/write accessible
  - Event buffers can be read
Event/interrupt support in OR10N

- **Event support on core side:**
  - **Enter sleep mode:**
    - Flush pipeline with `lpsync` instruction in order to enter a stable state
  - **Wake up:**
    - When an event is received continue in program flow
- **Interrupt support on core level:**
  - When an interrupt is received, the PC is stored in the exception PC register (EPCR)
  - PC is switched to address of interrupt exception
  - All registers are saved on stack
  - Jumps to global interrupt handler
    - Checks its interrupt vector table to select the correct interrupt handler
    - Up to 32 interrupt handler are supported
  - Jumps to emergency interrupt routine if high prio interrupt is received
    - Only one handler for fast response
  - Nested interrupts are not supported

- **Exception controller tasks:**
  - Flush pipeline when going to sleep
  - Store PC in EPCR
  - Set NPC to EPCR when return from exception instruction (`l.rfe`) is decoded
  - Wake up core controller
OpenRISC exercise session:

• In the exercise we are going to cover:
  
  – How to compile and run an application using:
    • The instruction set simulator (ISS)
    • The RTL simulation platform
  
  – Impact of the new instructions:
    • Hardware loops
    • Pre/post increment
    • Vector support
  
  – Interrupt and event handling