**RISC-V**

**A state-of-the-art open-source RISC with potential for embedded applications**

**RISC-V features**

- **Small CPU**
  - May have 16 registers,
  - Basic version includes only 33 instructions

- **Ideas borrowed from SPARC**
  - Register 0 is a constant 0
  - Presence of instruction register, gives rise to a simple, two-stage fetch-execute pipeline

- **Ideas borrowed from MIPS**
  - No status register
  - Conditional branches evaluate branch condition in the same cycle
  - Memory-mapped I/O

- **Novel ideas**
  - ALU intentionally lacks condition codes, branch conditions are evaluated by separate logic
  - RISC-V does not define or flag most arithmetic errors
  - Extra space is reserved for new instructions, future extensions but also user defined instructions

**Origins of RISC-V**

- Very recent
  - New instruction set architecture (ISA) originated in 2010 at University of California, Berkeley.
  - Gained momentum in the last 2 years
  - RISC-V v2.0 ISA specification is dated 6 May 2014
  - [http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf](http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf)

- Small integer ISA
  - A base for customized accelerators and extensions
  - 32-bit, 64-bit, 128-bit address space variants
  - Support for highly-parallel multicore and manycore implementations, including heterogeneous multiprocessors
  - Optional variable-length instructions
  - Freely available CPU designs, under BSD licence
    - BSD – Berkeley Software Distributions
    - The licence allows derivative works to be either open or close and proprietary
    - By contrast, ARM and Imagining charge significant licence fees and require NDAs (Non-disclosure agreements)

**Article in EE Times, 7 Aug 2014**

- Krste Asanović and David Patterson, UC Berkeley, "The case for RISC-V"
- Both authors are the originators of RISC-V
- Their argument pro RISC-V
  - New approach to ISA: base plus extensions, small core set of instructions
  - Enhanced floating point: quadruple precision as well as single- and double-precision.
  - Enhanced data bus, 128-bit as well as 32- and 64-bit
  - Comparison with other open core RISC designs:

<table>
<thead>
<tr>
<th>ISA</th>
<th>Complex Floating Point</th>
<th>64-bit</th>
<th>128-bit</th>
<th>128-bit Low</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARC V8</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>OpenRISC</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
<tr>
<td>RISC-V</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
<td>☑</td>
</tr>
</tbody>
</table>
If the branch was predicted not-taken, and it was actually taken (i.e. a loop was entered), then table Bpred is updated by adding the appropriate PC+4 and branch or jump target.

If the branch was predicted taken and it was actually not-taken (i.e. a loop exits), then there is no need to update the table.

The predictor uses a four entry table Bpred. Each entry is a <pc+4,target> pair. In the fetch stage, the predictor uses PC+4 to index the table. The predictor reads the corresponding <pc+4,target> pair from the table, and if the PCs match then this is a hit. If the PCs do not match, then this is a miss and PC+4 is used as the next PC instead of target.

The hit/miss signal is pipelined to the execute stage. This signal tells the control logic if the branch was predicted taken or not taken. Actions of the control logic are:

Predicted | Actual | Mispredict? | Action to take
----------|--------|-------------|----------------
taken      | taken  | yes         | no action required
not-taken  | taken  | yes         | fill last in fetch, update table, pc := branch or jump target
not-taken  | not-taken | yes      | fill last in fetch, do not update table, pc := current pc+4
not-taken  | not-taken | no       | no action required

R-type: rd <- rs1 op rs2 (register only instructions)
I-type: rd <- rs1 op imm (register and immediate)
S-type: memory store instructions; Loads are implemented as I-type
U-type: instructions involving PC, e.g. JAL (jump and link) and other instructions involving large immediates, e.g. LUI (load upper immediate)
There are also "synthesised" instructions, e.g. NOP == ADDI x0,x0,0

• Extended instruction sets are also defined, e.g.:
  - RV64I, RV128I (64-bit and 128-bit integer instructions),
  - Extensions for integer multiplication and division,
  - Extensions for floating point (single, double and quadruple precision),
  - Extensions for bit manipulation,
  - Extensions for vector architectures (SIMD)
• xi, where i=0..32, is used in RISC-V assembler to represent register i
• Register x0 is always 0
• NOP is implemented as: ADDI x0,x0,0
• J is implemented as: JAL x0, address
  - JAL rd, address normally calls a subroutine, where PC+4 is saved in rd