Fast Interrupts

Krste Asanovic, UC Berkeley / SiFive Inc. (Chair)
Kevin Chen, Andes (Vice-Chair)

8th RISC-V Workshop
Barcelona Supercomputer Center, Barcelona, Spain
May 9, 2018
RISC-V for Embedded

- Embedded is one of major uses for RISC-V
- Many different kinds of embedded system:
  - single microcontroller with 1KiB SRAM and simple I/O
  - 32 cores of dual-issue superscalar with L1/L2 caches, PCIe
  - 1024 cores with scratchpads, local I/O per core
- Desire for faster interrupt handling with support for nested preemted interrupts
- Task Group started XXXX
- Early days, only a few meetings so far – please join!
Summary of Current RISC-V Interrupts

- Local Interrupts
  - Directly connected to one hart
  - No arbitration between harts to service
  - Determine source directly through xcause CSR
  - Only two standard local interrupts (software, timer)

- Global (External) Interrupts
  - Routed via Platform-Level Interrupt Controller (PLIC)
  - PLIC arbitrates between multiple harts claiming interrupt
  - Read of memory-mapped register returns source
Current RISC-V Interrupt Overview

Global Interrupts

External Interrupt 1
External Interrupt 2
PLIC

Local Interrupts

Hart 0
- U
- S
- H
- M

Hart 1
- U
- S
- H
- M

Timer
Software

Timer
Software

Timer
Software

Timer
Software

Timer
Software
### Machine Interrupt Pending (mip) CSR

(Add custom local interrupts here, bits 16 and up)

<table>
<thead>
<tr>
<th></th>
<th>MXLEN-12</th>
<th>MXLEN-10</th>
<th>MXLEN-8</th>
<th>MXLEN-6</th>
<th>MXLEN-4</th>
<th>MXLEN-2</th>
<th>MXLEN-0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WIRI</td>
<td>WIRI</td>
<td>SEIP</td>
<td>UEIP</td>
<td>MTIP</td>
<td>WIRI</td>
<td>STIP</td>
</tr>
<tr>
<td></td>
<td>MEIP</td>
<td>MEIP</td>
<td>SEIP</td>
<td>UEIP</td>
<td>MTIP</td>
<td>WIRI</td>
<td>STIP</td>
</tr>
</tbody>
</table>

- **mip** reflects pending status of interrupts for hart
- Separate interrupts for each supported privilege level (M/S/U)
- User-level interrupt handling ("N") optional feature when U-mode present (discussed later)
Machine Interrupt Enable (mie) CSR

<table>
<thead>
<tr>
<th></th>
<th>MEIE</th>
<th>WPRI</th>
<th>SEIE</th>
<th>UEIE</th>
<th>MTIE</th>
<th>WPRI</th>
<th>STIE</th>
<th>UTIE</th>
<th>MSIE</th>
<th>WPRI</th>
<th>SSIE</th>
<th>USIE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MXLEN-12</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MXLEN-1</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>

- **mie** mirrors layout of **mip**
- provides per-interrupt enables

- External from PLIC
- Local Timer
- Local Software
Interrupts in mstatus

- Only take a pending interrupt for privilege mode \( x \) if \( x \cdot \text{IE} = 1 \) and running in mode \( x \) or greater
- Interrupts always disabled for privileges less than current level

Per-privilege global interrupt enables

Enable stack

Privilege stack
Interrupt reported in mcause CSR

<table>
<thead>
<tr>
<th>MXLEN-1</th>
<th>MXLEN-2</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt</td>
<td>Exception Code (WLRLE)</td>
<td>MXLEN-1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- High bit of mcause is 1
- Exception code gives interrupt cause
- Might have to interrogate PLIC also, if external interrupt
Two modes:
- Direct - all exceptions and interrupts jump to base
- Vectored – exceptions to base, interrupts vectored
**Vectored Interrupts (mtvec mode=1)**

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>Exception Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>User software interrupt</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Supervisor software interrupt</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td><em>Reserved</em></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>Machine software interrupt</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>User timer interrupt</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>Supervisor timer interrupt</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td><em>Reserved</em></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>Machine timer interrupt</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>User external interrupt</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>Supervisor external interrupt</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td><em>Reserved</em></td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>Machine external interrupt</td>
</tr>
<tr>
<td>1</td>
<td>≥12</td>
<td><em>Reserved</em></td>
</tr>
</tbody>
</table>

- Exceptions jump to: BASE
- Interrupts jump to: BASE+ExcCode*4
- Interrupts disabled at entry to handler, i.e., \texttt{mstatus.mie}=0

12-15 reserved
16+ for custom local interrupts
Saving/Restoring Interrupted Context

- **mepc** holds PC of interrupted instruction
- **mstatus.mpp[1:0]** in mstatus holds privilege mode of interrupted instruction
- **mstatus.mpie** holds interrupt enable of interrupted instruction
- Restored at end of handler by an **mret**
Problems with Current Interrupts

- Only hardware preemption is via privileged modes
  - Each privilege mode has independent hardware xepc and xpp/xie to save interrupted context
- Fixed priority for local interrupts, but fast vectoring
- Vector table holds jump instructions, can only jump +/- 1MiB.
  - Need free register to jump further, more instructions/vector entry
- PLIC has variable priority, but no vectoring
- PLIC need two memory accesses to claim/complete
- Unix ABI requires many registers to be saved/restored
Input from Multiple Proposals

- Andes enhanced PLIC
  - adds stacked preemptive servicing and vectoring to PLIC
- Syntacore
  - Restructure existing *IP/*IDELEG to provide faster interrupts on one core using existing local interrupt vectoring scheme, preemption via privilege mode
- Seagate
  - Add PLIC-based vectoring
- Liviu Ionescu
  - Multiple interrupt levels, vector table holds C-function addresses, hardware stacking with new embedded ABI
- SiFive
  - Multiple nested levels/priv mode, plus accelerated software save/restore
Interrupt Handler Interfaces

- Regular C function:
  - Interrupt hardware or software trampoline has to save/restore all caller-save registers

- Inline handler:
  - Use gcc interrupt attribute to convert function to always callee-save every register (save as you go).

- Opinion: Both are needed. C provides convenient interface, but even with hardware support adds unnecessary overhead for short handlers.
void attribute ((interrupt))
foo2 (void)
{
extern volatile int INTERRUPT_FLAG;
INTERRUPT_FLAG = 0;

extern volatile int COUNTER;
#elif __riscv_atomic
__atomic_fetch_add (&COUNTER, 1, __ATOMIC_RELAXED);
#else
COUNTER++;
#endif
}

foo2:
addi sp, sp, -FRAMESIZE  # Create a frame on stack.
sw s0, offset(sp)      # Save working register.
sw x0, INTERRUPT_FLAG, s0  # Clear interrupt flag.
sw s1, offset(sp)     # Save working register.
lw s0, COUNTER      # Get counter address.
lw s1, 1
amoadd.w x0, (s0), s1 # Increment counter in memory
lw s1, offset(sp)    # Restore registers.
lw s0, offset(sp)
addi sp, sp, FRAMESIZE     # Free stack frame.
mret
Unix C ABI has high context switch cost

The current RISC-V ABI requires (ignoring floating-point registers):

```assembly
addi sp, sp, -FRAMESIZE
sw ra, x(sp)    # 1 return address
sw t0-t6, x(sp) # 7 temporaries
sw a0-a7, x(sp) # 8 arguments
jal c_handler

lw a0-a7, x(sp) # 8 arguments
lw t0-t6, x(sp) # 7 temporaries
lw ra, x(sp)    # 1 return address
addi sp, sp, FRAMESIZE
mret
```

or 36 instructions for each interrupt vector. Not including floating-point. Putting save/restore in hardware won’t help much, need new ABI.
Vectoring Options

- Current PC jump constrains range
- Put function pointers in table (Seagate, Andes, Liviu)
  - Hardware has to fetch pointer and jump to it
- Add new +/- 2GiB offset instruction only visible in new interrupt ISA mode (SiFive)
  - Looks like single regular instruction to pipeline
  - Entry contains xxxxxxxxxxxxxx011, 32-bit instruction with new PC relative addressing mode
- Want vectoring for hardware save/restore or inline ISRs, but no vectoring for software trampoline
Interrupt Mode, Level, Priority

- Each interrupt is configured with:
  - Privilege mode: M/S/U where supported
  - Interrupt level: 1-N, N is number of preemption levels
  - Priority: is priority within a single preemption level (doesn’t preempt but serviced in priority order)

- Various schemes to allow fixed number of bits to be allocated dynamically among these fields

- Single max reduction in interrupt controller picks highest privilege/level/priority to service next
Managing Preempted Context state

- Push/pop mepc/mpp/mil/mie to memory stack using hardware or software
  - Now need to remember previous interrupt level, mil, also
- Add additional mepc/mpp/mil/mie registers in hardware registers
- Pack mpp/mil/mie into mcause to simplify save/restore and reduce number of CSRs
Other issues

- Handling multiple privilege levels, security and stacks
- Quickly servicing remaining interrupts
  - Go from exit of one handler to entrance of another without restore/save again
- Supporting large numbers of MSI interrupts
  - Interaction with virtualization
Foundation Fast Int TG Work Meeting

- Thursday 3:30-4:30pm, Track 1