John Hauser’s
Stacked Traps for RISC-V

Preface

- The design presented here is still a work-in-progress. Improvements and fixes are possible with good reasons.
- This presentation is an overview, leaving out some details. More complete specifications can be provided (and refined) if there is interest.

Main Targeted Features

- Vectored interrupts
  - Handler address is read from table, indexed by interrupt number
- Automatic nested interrupts (preemption)
  - Based on priority level, of course
- Option for using standard-ABI subroutines as handlers
- Option for faster interrupt handlers
  - Using special calling convention instead of standard ABI

Assumptions about Memory

- Vector tables will be in “local memory”, fast to access
  - Also known as “tightly coupled memory”, other names
  - Harts are allowed and expected to enforce this, by constraining table locations; details are mostly implementation-specific
- Interrupt stack expected also likely in local memory, for fast register saves
- Inconvenient access faults (e.g. PMP violations) can be treated as abnormal, raising a double-trap exception

Additional Observations

- Easiest if same trap entry/exit machinery used for both interrupts and synchronous exceptions
  - Also, baseline RISC-V exception entry/exit *blocks* interrupts; want to minimize this blockage, same as for interrupt trap entry/exit
- Mechanism for taking traps can be decoupled from interrupt identity hierarchy
  - My plan is to combine stacked traps with AIA interrupt hierarchy
  - Could instead do stacked traps with a flat interrupt hierarchy
  - Different interrupt hierarchy implies different vector table structure

Baseline Trapping vs. Stacked Traps

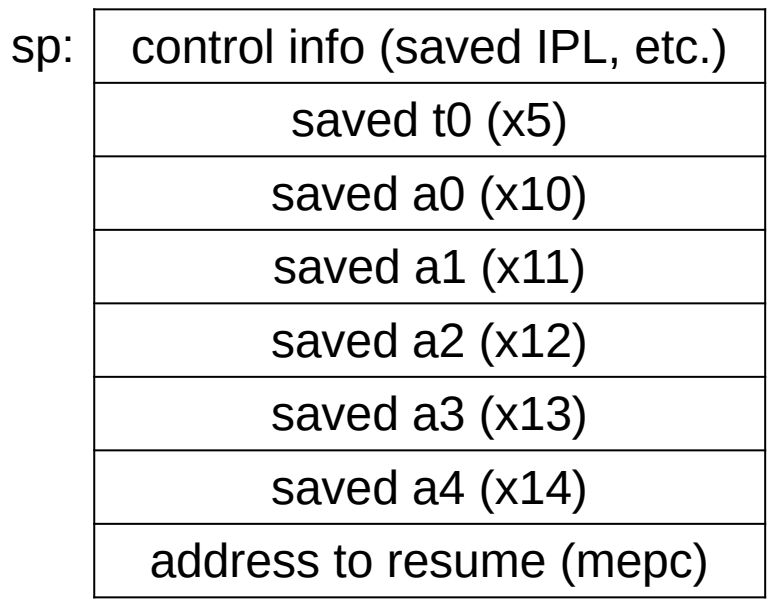
- "*Baseline trapping*" is the original RISC-V trapping mechanism
- The trapping mode (baseline or stacked traps) is selected by low bits of mtvec register

Stacked Traps, Part 1: Fast Handlers

Automatic on trap entry:

- The stack (sp) is swapped as needed
- A small trap stack frame is pushed on new stack (→)
- IPL is set to new priority level
- Handler's entry address is read from table in local memory

<... next slide>



4 bytes (RV32) or 8 bytes (RV64)

Stacked Traps, Part 1: Fast Handlers

<... continued>

- Registers a0-a3 are written with trap information:

	synchronous exception	interrupt, not external	interrupt, external
a0	mcause	mcause	mcause
a1	mtval	interrupt priority	minor identity and interrupt priority
a2	mtval2 or 0	0	0
a3	mtinst or 0	0	0

- Last, execution jumps to handler’s entry address

Stacked Traps, Part 1: Fast Handlers

MRET is still used for trap exit:

- Stack frame is popped, restoring registers (including IPL)
- The stack (sp) is swapped back, as needed
- Execution jumps to resume address (read from stack frame)

Stacked Traps, Notes

- Any fault during automatic reading of vector table or pushing/popping stack causes a double-trap exception
- Interrupts are not disabled by trap entry, and can stay enabled up to trap exit (MRET).
  - Immediately after trap entry, a preempting interrupt can be taken
- Some CSRs for baseline trapping are ignored (must be ignored!) by software:

  mepc, mcause, mtval, mtval2, mtinst

Implementation Optimizations

- Can have dedicated fast path for automatic read of vector table in local memory
- Pushing stack frame and writing a0-a3 can overlap with pipeline redirect on trap entry
- MRET can sometimes shortcut next trap entry when another interrupt is pending
  - Leave stack frame alone and just write new trap info into a0-a3, before jumping to new handler entry address

Latency to Enter a Fast Handler

- **IF** each value pushed on stack takes 1 cycle,

  And **IF** all other automatic actions on trap entry completely overlap stack pushes,

  → Time to enter fast trap handler is 8 cycles
- With a short pipeline and wider stores to memory, possible to conceive of faster implementations

  (4 cycles?)

Main Targeted Features

- Vectored interrupts
  - Handler address is read from table, indexed by interrupt number
- Automatic nested interrupts (preemption)
  - Based on priority level, of course
- Option for using standard-ABI subroutines as handlers
- Option for faster interrupt handlers
  - Using special calling convention instead of standard ABI

Stacked Traps, Part 2: Standard-ABI Functions as Handlers

- Bit 0 of handler entry address (from vector table) selects one of two paths:
  - If bit 0 = 1: fast handler, using nonstandard calling convention, as in earlier slides (only t0 and a0-a4 saved automatically on stack)
  - If bit 0 = 0: standard-ABI handler
- If bit 0 = 0, differences on trap entry:
  - *Initial* handler address is taken from mtvec, not from vector table → single common entry for all standard-ABI handlers
  - Handler address from vector table is automatically written to t0

Stacked Traps, Part 2: Standard-ABI Functions as Handlers

- Initial handler from mtvec is a “fast handler” that prepares for and calls the standard-ABI handler

common initial handler entry:

... (optional) Push caller-saved F registers, others

... Push remaining caller-saved X registers (**cm.pushtx**?)

`li gp,___` or `lw/ld gp,___`

jalr t0 @ Call standard-ABI handler from vector table

mchaini @ Chain any pending interrupts

... Pop registers saved earlier (**cm.poptx**?, ...)

`mret`

New Instruction: MCHAINI

- “Chain Interrupt”
- If no interrupt trap pending, does nothing (no-op)
- Else, can JAL to new handler, reusing same push/pop of all registers
  - Leaves ra pointing to MCHAINI, so on return from trap handler, gets executed again
  - (Special handling if new trap handler is a fast handler)

```
initial entry:
    ... Push registers
    ... Prepare gp
    jalr t0
»   mchaini
    ... Pop registers
    mret
```

New Instruction: MCHAINI

- For simpler implementations, allow MCHAINI = no-op?
  - Loses ability to chain standard-ABI interrupt handlers
  - But no loss of functionality

```
initial entry:
    ... Push registers
    ... Prepare gp
    jalr t0
»   mchaini
    ... Pop registers
    mret
```

Implementation Optimization

- On trap entry for a standard-ABI handler (bit 0 = 0), automaticaly push handler address on return address stack, so JALR target can be predicted correctly here

```
initial entry:
    ... Push registers
    ... Prepare gp
»   jalr t0
    mchaini
    ... Pop registers
    mret
```

M-Level Option: Only Stacked Traps

CSRs for stacked traps:

mtvec	trapping mode + main trap vector
mscratch	holds stack pointer for trap handlers
mtctl	Trap Control
mtvt	Trap Vector Table - synchr. exceptions and major interrupts
meitvt	External Interrupts Trap Vector Table - AIA external interrupts
mipreempt	Interrupt Preemption configuration - “priority levels”
mipl	current Interrupt Preemption Level
miscratch ?	alternate stack pointer for high-priority interrupts (optional?)

CSRs eliminated:

mepc, mcause, mtval (mtval2, mtinst)

M-Level Option: Baseline + Stacked Traps

- Support both modes for greater software compatibility

  CSRs used for all trapping modes:

  mtvec, mscratch

  Only for baseline trapping:

  mepc, mcause, mtval (mtval2, mtinst)

  Added CSRs (some usable with baseline trapping too):

  mtctl, mtvt, meitvt, mipreempt, mipl, miscratch

Stacked Traps at S-Level

- Stacked trapping possible also at S-level, but ...

  Does this make sense?

- Executing anything in M-mode totally blocks S-level interrupts

  How many clock cycles? N = 500? 5000?

- Cannot guarantee interrupt latency better than N

  Is 500 + 8 cycles that much better than 500 + 20?

Further Topics

- ... Many details
- Vector tables structure
- Configuration of preemption cohorts (“priority levels”)
- Additional features:
  - “Context interrupts”: High-priority interrupt handler can schedule lower-priority task as another interrupt
  - Priority deference from M-level to S-level: High-priority S-level interrupt is taken before low-priority M-level interrupt