Zephyr™ Project
Developer Summit
June 8-10, 2021 ▪ @ZephyrIoT

Weaving Xtensa Yarns with Zephyr

NOTES ON AN IDIOSYNCRATIC ARCHITECTURE

Zephyr™ Project
Developer Summit
June 8-10, 2021 • @ZephyrIoT

Andy Ross

SOFTWARE ENGINEER – INTEL

andrew.j.ross@intel.com
andyross @zephyrproject.slack.com
andyross @github

Zephyr on Xtensa

- Highly Configurable CPU IP from Tensilica, now Cadence
- Supported in Zephyr since 2018
  - Espressif ESP32
  - Intel Audio DSP Family (2013 onwards)
  - ODroid Go (Gameboy Emulator)
- Natural fit for applications with ISA extension needs
  - Vector DSPs
- Licensed as customer-configured IP only
  - No single "CPU" designs

Xtensa Instruction Set Architecture

Zephyr™ Project
Developer Summit

- Traditional pipelined RISC
  - Load/store architecture
  - Single-issue
  - In-order execution
- 16 General Purpose Registers
  - A0-A16
  - A1 is stack pointer by convention
- 32 interrupts vectored to configurable (e.g. 7) number of priority levels
  - Fixed priority for each interrupt
  - Level 1 shared with exception processing

Xtensa ISA, weirder stuff

- Configurable "State Registers"
  - Move to/from GPRs with RSR/WSR instructions
  - Named with one byte in the instruction, space of 256 possible SRs
  - Easily enumerable by the OS for context management
    - (Mostly, the assembler makes this harder than it has to be)
    - Existence exposed by system headers (core-isa.h)
  - Examples:
    - PS – Program state register
    - LCOUNT/LBEG/LEND - LOOP instruction extension
    - SCOMPARE – Atomic test/set instruction operand
    - SAR – Shift and rotate instruction operand
- One toolchain per instantiated CPU design!

Xtensa ISA, weirder stuff...

- Register Windows!
  - 16 visible GPRs, but 16, 32 or 64 actual registers
  - Rotated in groups of 4 registers upon function call
    - Call can "save" 4, 8 or 12 of its own registers
      - (Or zero, which is a variant "CALL0" ABI for configurations with only 16 registers)
  - Cyclic with spill/fill managed by exception handlers
    - Trying to return to registers that are not present traps to fill them from the stack
    - Trying to use a GPR in a register block "owned" by an upthread function traps to spill
      - Trap happens at use time, not call time!
  - Faster function calls, in theory
    - In practice, generated code tends to be poor
    - "How many registers to use" is a global optimization (depends on callee behavior)
    - But register assignment happens locally
    - Upshot: code uses way too much stack for calls
    - Xtensa's proprietary compiler backend does a little better than gcc
  - Details very clever, but very complicated

Zephyr Interrupt Handling

Zephyr™ Project Developer Summit

- Hardware interrupt entry is mostly conventional
  - Interrupts jump automatically to one of N handlers
  - Mask interrupts unconditionally
  - Set EPCn to the interrupted program counter, EPSn to the interrupted state register
  - Don't touch GPR or window state
  - Provides EXCSAVEn for use as a scratch register
    - Zephyr doesn't use it!
- Saving context
  - We have to switch stacks (Zephyr has a separate interrupt stack)
  - The register windows assume a single stack
  - Oh well, must spill all 64 GPRs on interrupt
  - Or must we...

Zephyr Cross Stack Calls

- Carefully construct a first call frame containing the interrupted stack frame
  - Will be restored on return when unspilled
- Push a second 4-register frame
  - Acts as a spill area for the new stack that won't clobber the first
- Forcibly set the stack pointer (old one will be restored on return)
- Call the interrupt handler (which is a standard C function)
- No registers need to be explicitly spilled at all!
  - Everything happens via natural windows exceptions
  - Interrupt entry overhead is on the order of two nested functions
  - If the interrupt handler is shallow, interrupted GPRs may never be spilled at all
  - Takes most overhead on interrupt exit, not entry

Interrupt Handling

Zephyr™ Project
Developer Summit

- One C function to handle each interrupt level
- Multiple interrupts per level
  - (Also level 1 includes exceptions)
- Of the 32 interrupts, each belongs to exactly one level
- Generate optimized C code with `xtensa_integen.py`!
  - Bit test only the interrupts that are possible at each level
  - Call the handler out of the Zephyr isr table
- Not as great as I would have hoped...
  - Function call overhead is expensive
  - Xtensa bit handling surprisingly slow
  - Most hardware turns out to have a second level interrupt controller anyway

Interrupt Exit

- Call `z_get_next_switch_handle()` to find next context to run
- Common case: returning into same thread
  - No need to unspill windowed registers, just the interrupted GPRs
  - Even those will only have spilled the registers that were in use
  - Restore EPCn/EPSn and issue RFI to return
  - Zephyr RFI path is shared, always uses the same (highest) level for RFI
- Uncommon case: preemption
  - Leave the spilled interrupt context on the stack in place
  - But all the caller window frames need to be spilled! ...

Zephyr Window Spill

Zephyr™ Project
Developer Summit

- Provided by HAL layer
  - Big (few hundred bytes of code)
  - Software-only solution, interprets window state.
- Zephyr wrote our own
  - Relies on exception handlers
  - Just poke registers to spill them
  - Tiny (-------> !!!)
  - Actually about twice as fast!
  - (Hardware exception processing is very speedy, use hardware)
- Also useful for exception handling
  - Debugger integration

```
/* And this is it! */
.macro SPILL_ALL_WINDOWS
#if XCHAL_NUM_AREGS == 64
        and a12, a12, a12
        rotw 3
        and a12, a12, a12
        rotw 3
        and a12, a12, a12
        rotw 3
        and a12, a12, a12
        rotw 3
        and a12, a12, a12
        rotw 4
#elif XCHAL_NUM_AREGS == 32
        and a12, a12, a12
        rotw 3
        and a12, a12, a12
        rotw 3
        and a4, a4, a4
        rotw 2
#endif
.endm
```

Symmetric Multiprocessing

- Xtensa cores often deployed in SMP configurations
  - Shared memory bus, same address space
  - ESP32: two cores, second dedicated to network in upstream use cases
  - Intel Audio DSP: 2-4 core configurations shipping
  - No MMU (yet)
- First Zephyr SMP Platform
  - Prototyped most of the SMP APIs
  - Made most of the early mistakes

SMP Requirements

- "What CPU am I?"
  - Kernel needs fast (!) access to per-CPU struct
    - Contains _current thread pointer
  - Use a dedicated SR
    - Must be "priviledged" (for when we get an MMU) so user code can't mess it up
    - Not saved by context management, set once at boot
    - No architecture-specified register for this
  - Indirect via `CONFIG_XTENSA_KERNEL_CPU_PTR_SR`
    - On esp32, uses "MISC0", which is an excellent choice
    - No MISC on Intel, use EXCSAVE2 instead
    - Remember Zephyr doesn't use it! Nice to be frugal.
  - Maybe SR should store the thread pointer instead?

SMP Requirements...

- "Start CPU n"
  - No architectural interface for this at all.
  - ESP32 provides a HAL call
  - Intel DSP does this via the x86 host CPU!
    - Requires driver protocol, working this out with SOF right now
- "Interrupt other CPUs" (`arch_sched_ipi()`)
  - Needed for scheduling, to inform CPUs of changes to the set of runnable threads
  - Again, no architectural interface
  - Intel DSP has custom "IPC" (Inter-processor communication) hardware
    - Doorbell call/response registers
  - ESP32 just can't do it at all
    - At least not via a documented API

SMP Requirements...

- Atomic memory access API
  - Core primitive from which spinlocks are built
  - Generally built on top of a single instruction "compare and swap"
  - Which xtensa has: `S32C1I`
  - But the compare operand lives in the SCOMPARE SR!
    - Led to a subtle bug: atomics are usually used in spinlocks, but not always
    - If SCOMPARE isn't saved with context, we can be interrupted before its used!
    - We went 2 years without SCOMPARE saved in context

L1 Cache

- Xtensa often used in shared/slow/contended memory environments
- For example, the Intel Audio DSP:
  - Has two SRAM regions, one slower (and lower power) than the other
  - Also system DRAM access, which is VERY slow
  - Also a "TLB" translation lookup interposed in front of the SRAM access eating cycles
- So the CPU can be configured with L1 data and instruction caches
  - Typical 64 byte cache line size
  - Configurably N-way set associative
  - Configurable size, e.g. 16 kb
  - Straightforward writeback/flush/invalidate control via dedicated instructions
  - **AND IT IS INCOHERENT**

Incoherent?

- Cache incoherence breaks basically all memory reasoning
  - Changes to memory from CPU0 stay in cache and aren't visible to CPU1
  - Changes from CPU1 can get flushed at any time and clobber CPU0 unexpectedly
  - Changes to "unshared" memory merely in the same cache line can be clobbered too
  - Even uncolliding changes may appear to other CPUs in any order
  - **EVERYING** in the kernel assumes universal visibility
- How to fix this?
- Idea #1: Don't
  - Disable the cache
  - Guarantees correctness at expense of performance
  - Works well if SRAM is fast relative to CPUs. ESP32 does this
  - Intel ADSP takes 12 cycles to read from SRAM!

Managing Incoherence

- Idea #2: Harden the kernel
- Make all kernel code incoherence safe.
  - We already know when synchronization (spinlocks) happens, so leverage that:
  - Flush and invalidate the whole cache when lock is taken
  - Flush when lock is released
- This makes non-synchronizing CPU-bound code fast
  - But it does nothing for the kernel itself, which is actually much slower than Idea #1
  - Flushing the whole cache takes 256 instructions if done naively
- Still does nothing for the poor application, which has all the same issues

Managing Incoherence

- Idea #3: CONFIG_KERNEL_COHERENCE
- Note one hardware trick:
  - The Xtensa memory protection option allows associating different cache behavior with different 512MB memory regions
  - The Intel ADSP maps addressible RAM twice, once cached and once uncached
- Recognize that almost all kernel data is pervasively shared
  - CPU structs, Thread records, Timeout objects, IPC primitives
  - All are designed to be used from any thread at any time
  - They really can't be cached anyway
  - Put them in uncached memory!

KERNEL_COHERENCE...

- Recognize that almost all stack accesses are intended to be unshared
  - And the stack makes up 80%+ of memory accesses in common code!
  - Put the stack in cached memory
- Do all this in the linker.
  - `__coherent` and `__incoherent` attributes for user-defined symbols
  - Make default data coherent (uncached) by default, for correctness
- Add runtime checks to catch errors
  - Code may want to place e.g. IPC objects on the stack, and it's legal elsewhere!
  - Means I have to play "coherence police" with regressions in the test suite

KERNEL_COHERENCE...

- What about context switch? This gets complicated
  - Must spill register windows first! (remember those?)
  - Need to flush out the old thread's stack, it may not run on this CPU next
  - Need to invalidate the new thread's stack, we may not have been the last to run it
  - Don't need to flush the dead area below the top of the old thread, it's not used
  - Don't forget the caller's spill region below the current stack pointer!
- This gets handled by `arch_cohere_stacks()` inside `z_swap()` and interrupt exit
  - But we can't do it all in C, because we're still running on the old stack.
  - Does most of the logic, but leaves the flush to assembly called immediately before switch
  - Stashes the computed pointer in EXCSAVE3 (useful again to have extra registers)

KERNEL_COHERENCE...

- Future optimizations:
- Some things in kernel data are unshared, or read-only, or change rarely
  - Ex.: CPU ID number never changes
  - Thread priority changes require an IPI and are heavyweight anyway
  - IPC configuration data (semaphore max size) only change at initialization
- Would be nice to have a framework to separate these

Memory Management

- Current Zephyr Xtensa platforms don't provide MPU/MMU hardware
  - Yet...
- Cadence offers both as options
  - Both are "oddly clever" as is Xtensa custom
  - Both are good targets for Zephyr userspace implementations
  - Neither are implemented in Zephyr
    - Yet...
- Xtensa MPU:
  - No address translation
  - 4k aligned region boundaries
  - 16 or 32 region registers partition all 4G of memory space
  - Means there are 8 or 16 possible "useful" regions, plus one for each two that are adjacent

Future: Memory Management...

- Xtensa MMU
  - Conventional paged memory translation unit, mostly
  - Read/write/execute and cacheability control per page
  - Set-associative TLB with configurable size
    - Also some special TLB ways with specific behavior for e.g. pinning and large page size
  - Software exception used to fill TLB on a page table miss
  - Hardware engine fills TLB from page table normally
  - Page table is a linear array in **virtual address space**
    - Means it has to map itself
    - But also means hardware has no special paths, or state machine
    - Software handling (per above) handles bootstrapping of paging

Future: Zephyr Userspace Requirements

- Syscalls: already have exception code dedicated, and instruction to trigger
  - Just need to decide on marshalling scheme for arguments
- Interrupts: Can't rely on interrupted user stack
  - Might be too small, and fault (!USERSPACE just overflows)
    - Might also be deliberatedly misconfigured with a kernel address!
  - Must switch stacks before context save
  - Complicates cross-stack calls
    - Maybe do only when kernel threads interrupted?
- Security: need to scrub kernel registers in windows
  - Not all Zephyr archs do this currently
  - Force spill and zero? (similar code to above, with writes)

“It is one of man's curious idiosyncrasies to create difficulties for the pleasure of resolving them.”

- Joseph de Maistre

Zephyr™ Project

Developer Summit

June 8-10, 2021 ▪ @ZephyrIoT