Lecture 7, Feb 27, 2024

Memory Mapping

- In a *unified memory space*, the code, data, and stack are together, and the compiler can arrange these easily to optimize
- In embedded systems, we often have *separated memory spaces*
  - Compilers have a hard time understanding and optimizing these, because some sections of memory can support different types of access, have different speeds, etc
  - We may need to e.g. explicitly tell the compiler to put something in non-volatile memory so it doesn't take up RAM
- Typically external memory cannot be utilized automatically by the compiler
  - Some microcontrollers may have DMA to access these, but compilers doesn't understand these
  - The way to do this typically varies per-chip and per-compiler
- Therefore we need to understand how the compiler does low-level optimization (what it does and doesn't optimize) to write good high-level code

Optimization and Compilation

- For embedded systems, compiler toolchains are often a lot less developed than for desktop architectures, so we need to be mindful of writing optimized or easy-to-optimize code
- We usually optimize for one of two metrics: execution time and program space
  - Often faster code is shorter code, but this is not always true (e.g. unrolling a loop)
- Optimizing code too heavily will lead to unreadable and unmaintainable code
  - Only optimize timing-critical or heavily reused/iterated code
- There are 3 common forms of optimization:
  - *Global optimization*: making algorithmic changes to the code over multiple lines
    - This is high-level (before code generation) and often difficult for compilers
  - *Local optimization*: making optimizations within a single line/expression/statement
    - Even basic compilers generally do this well
    - We are not expected to do this by hand
  - *Keyhole/peephole optimization*: runs on the final assembly code and optimizes a few instructions at a time
    - Uses a sliding window and tries to match known optimizations
    - No human intervention
    - When two templates meet, there can be inefficiencies, e.g. storing a register into memory only to load it back again; this will be optimized away by keyhole optimization
    - Note that we can't always apply an optimization, due to e.g. interrupts, SFRs, DMA, etc
- Global optimization techniques:
  - *Loop unrolling*: expanding a fixed-length loop into repeated code
    - Increases code size but avoids overhead of looping
    - Unrolled code is less readable and harder to maintain
    - Compilers are often good at this, but only if the loop length is well-known (constant)
  - *Code motion*: factoring out unchanging code from inside a loop
    - e.g. if we're indexing a constant index of a constant array in a loop, we can move the indexing outside the loop and only do it once
    - Certain embedded compilers will be able to do this
    - Note that this uses extra temporary storage
  - *Strength reduction*: using loop structures to convert more complex/slower operations into simpler ones
    - e.g. if we're assigning `x[i] = c * i` in a loop over i, we can convert the multiplication to successive additions
    - Array operations are often a common source of these optimizations
    - Whether this is worth doing depends heavily on the platform and how fast each type of instruction executes

- *Common sub-expression elimination*: factoring out common sub-expressions that are used multiple times, and only doing it once
    - * Compilers do this pretty well
- *Lookup tables*: using pre-computed lookup tables instead of computing everything

- Local optimization techniques:
  - *Coalescence*: using the instruction set (side effects) to compile multiple operations
    - * e.g. `x = x + y` would normally be 4 instructions (2 loads, 1 add, 1 store) but can often be optimized to just one
  - *Constant folding*: pre-calculating constant values
  - *Local strength reduction*: strength reduction within a line
    - * e.g. multiplication by powers of 2 to shifting, exponentiation to repeated multiplication, etc
  - *Machine idioms*: making use of specialized instructions on the microcontroller
    - * e.g. counting down instead of up in a loop if comparison with zero is faster

Interrupts

- *Interrupts* are events that asynchronously affect the program flow
  - Calls to the *interrupt service routine* (ISR) are done automatically in response to the *interrupt source*
  - The ISR is called like a regular function, but it can be called at any time in the program flow
- Interrupt sources are often external, but we can have internal software-generated interrupts (SWI) as well
- The ISR is a special subroutine executed on an interrupt; since it can be called at any point during program execution, it must:
  - Make no assumptions about program or microcontroller state (e.g. register contents)
  - Make no (unexpected) modifications to the microcontroller state on exit
    - * Back up all registers and SFRs that we use
    - * Some architectures back up registers automatically on the stack
    - * Returning is usually done via a specialized return from interrupt instruction
  - As fast and short as possible, and have a deterministic exit condition (i.e. won't hang)
    - * Avoid:
      - Extensive shared resource use
      - Calls to additional subroutines (which can use a lot of stack)
      - Waiting for hardware, polling, delays
    - * Use timeouts or failsafes
- Interrupts are typically not enabled by default, so we have to set them up first
  - In ASM we define or set jump points through an *interrupt vector table* or set of SFRs
  - In high-level code this is often done with function pointers, pragmas, etc
- The vector table is used by the CPU to look up the address of the ISR to jump to
  - Often we don't have a one-to-one mapping from interrupt source to ISR due to hardware cost
  - In the ISR we need to scan through and identify the source of the interrupt
    - * This can be costly for heavily overloaded ISRs, especially external interrupts
- Interrupts can come from a number of sources:
  - External interrupt lines (IRQ lines)
    - * Triggering can occur on positive or negative edge, level triggering, or user-defined
  - Peripheral events (e.g. timers, ADC, protocol peripherals)
  - Software interrupts (SWI)
    - * This can be used for exception handling, multi-tasking, debuggers, etc
- All platforms provide a method to *mask* interrupts, selectively enabling or disabling interrupts
  - However, even when interrupts are masked, they often still accumulate in hardware and will trigger once we restore interrupts
  - We usually disable interrupts of the same source while in the ISR for that source, since the ISRs themselves can be interrupted
  - This is done using a `CLI` instruction to clear masks (allow interrupts) and `SEI` instruction to set

masks (disable interrupts)
    * Modern systems will have SFRs per interrupt source

- Some critical interrupts cannot be masked (*non-maskable interrupts*, NMIs)
  - This is used for critical tasks like bootloaders, watchdogs, e-stops, etc
- If two interrupts occur simultaneously, or we have multiple interrupts waiting after clearing the mask, *interrupt priority* is used to determine which gets handled first
  - This is often configurable in modern microcontrollers through a table; older platforms have a fixed table
  - Note this completely ignores order of arrival; higher priority interrupts are always handled before lower ones
  - We need to be more careful with high-priority interrupts so they don't monopolize the CPU