Embedded Applications

COMP595EA
Lecture 04
Interrupts
The response problem

• Embedded systems have response constraints.
  – Also require asynchronous processing
  – GUIs share the same problem (solved with events)

• Interrupts are one solution for embedded systems
  – Difficult to program
  – Require thorough systems knowledge and more importantly code/algorithm mastery.
  – Introduce other problems
From Human to Machine

[Diagram showing the process from High Level Source Code through Assembly Language, Assembler, and Compiler to Machine Code Bits.]

Solitude and Loneliness
Assembly Instruction Sets

• Every different family of microprocessor = different instruction set.
  – Requires different assemblers
  – Different development tools

• Within a specific family all microprocessor models generally share the same instruction set
  – possible addition/subtract of specific commands
Registers

• Almost all microprocessor have registers
  – Used to perform all calculations
  – Used to set/store settings and parameters
  – Used to control program execution
  – User modifiable (With god-like consequences)

• Number and type of registers is dependent on model of microprocessor
Execution Control

• Almost all microprocessors share two important registers
  – Program Counter
    • Contains the address of the next instruction to execute
  – Stack Pointer
    • Contains the address of the top most memory position in use on the stack.
    • Very important for supporting subroutines calls.
      – Hence important for interrupt routines too!
Cont.

• accumulator register
  – Some microprocessors can only support the destination of calculation operations into an accumulator register.
  – Others allow calculation into registers or into memory addresses

• Jump Instructions / Conditional Jump Instructions
  – Allow the program counter to be modified
Stack Operations

- **Push**
  - Copies a register or other value onto the stack and changes the stack pointer register.

- **Pop**
  - Copies the top most value addressed by the stack pointer to a register or memory location and adjusts the stack pointer accordingly.
Subroutine Stack Usage

• subroutines (as in Fortran 77) make use of the stack.

• Instruction sets provide two support operations
  – CALL
    • Pushes the following instruction address onto the stack
    • Places subroutine start address into the program counter
  – RETURN
    • Pops the top stack value from the stack and places it in the program counter register.
Interrupts

- Interrupts begins with a hardware signal.
- Signal is caused by some other chip.
  - frequently I/O chips
  - Sensors
- Pins from these chips are asserted when the chip requires attention from the microprocessor
- Pin is electrically connected to an interrupt pin on the microprocessor.
Interrupts are subroutines

-Asserting the interrupt causes (nearly) immediate interruption of whatever the microprocessor was in the middle of doing.

-The microprocessor stops what it was doing and begins execution of an interrupt subroutine that the program has written.

-Basically act exactly like a CALL statement to the interrupt routine were the very next instruction.
  - Except the CALL happens implicitly
Interrupt Housekeeping

- In addition to the primary task typically handle housekeeping chores before and after:
  - resetting interrupts
  - saving context
  - restoring context
- Last command in interrupt subroutine: RETURN
- Just like a CALLeed subroutine but the CALL is implicitly done by the hardware.
The New Problems

• Sample Code
  MOVE R1, (iCentigrade)
  MULTIPLY R1, 9
  DIVIDE R1, 5
  ADD R1, 32

• Interrupt Routine:
  // read char from I/O chip
  // store value in R1
  // reset and return
Saving and Restoring Context

• A better routine
  PUSH R1
  PUSH R2
  // read char from I/O chip
  // store value in R1
  // reset I/O chip
  POP R2
  POP R1
  POP R1
  RETURN
Contexts

- Most microprocessors can only operate on data that is in registers (aside from MOVE)
- Unreasonable to expect interrupt routines to avoid modifying registers
- Save any and all registers that the interrupt subroutine uses. Restore them before returing
  - Known as “Saving Context”
- Use as few registers as possible.
Nefarious Bugs

• Failure to Save/Restore context can cause bugs or failures that are EXTREMELY difficult to diagnose.
  – unpredictable, unrepeatable
  – only occur when the stars align
  – Don't always produce the same fault.

• Programmers really need to understand the subtle interaction of their code.
Disabling Interrupts

- Some I/O/Support chips have input pins that prevent them from asserting interrupts.
- More often interrupts are prevented by disabling the interrupt service action in the microprocessor.
  - Instruction set provides for disabling all interrupts or selectively disabling interrupts.
  - Non-maskable interrupt pins cannot be disabled if provided
    - Used for select critical or disastrous events
Interrupt Priority

• Most interrupts have a designated priority.
  – disabling is accomplished through a threshold value.
  – interrupts with a lower priority than the threshold are ignored.

• Sometimes used in conjunction with interrupt disabling instruction or instruction mask capabilities.
Questions

• How does the hardware know where to jump to?
  – Fixed addresses (limited subroutine length?)
  – Address tables (fixed or specifiable location)
  – Dedicated registers (expensive?)

• Can an interrupt occur in the middle of an instruction?
  – No.
  – Exception: block move instructions. (Z80, x86)
Questions Cont.

• What if two interrupts occur simultaneously which gets handled first?
  – Based on priority

• Can an interrupt signal interrupt another interrupt routine?
  – Usually: Yes.
  – Some require instructions be added to the routine to allow interrupt nesting.
  – Some disable interrupts (x86) before int. subroutine.
Questions Cont.

- What happens if an interrupt occurs while interrupts are disabled?
  - It gets ignored.
  - Some microprocessors remember interrupts others forget them if the interrupt line stops being signaled.

- What happens if you forget to reenable interrupts?
  - Nothing. (Literally)
  - Microprocessors are usually mostly about handling interrupts.
Shared-Data Problem

- Interrupts cause a problem because they share data with the non-interrupt (task) code.
- This results in problems similar to those found in concurrent code.

```c
    void interrupt vReadTemperatures(void)
    {
        iTemperatures[0] = // read temp1 data
        iTemperatures[1] = // read temp2 data
    }
```
void main(void)
{
    int iTemp0, iTemp1;
    while (TRUE)
    {
        iTemp0 = iTemperature[0];
        iTemp1 = iTemperature[1];
        if (iTemp0 != iTemp1)
            Detonate!!
    }
}
Fixing the problem (maybe)

• Can we fix the problem with:
  • if (iTemperature[0] != iTemperature[1])

• Nope! Compilers produce...
  • MOVE R1, iTemperature[0]
    MOVE R2, iTemperature[1]
    SUBTRACT R1, R2
    JCOND ZERO, TempOK
    Detonate!
  TempOK:
  ...
  ...
  ...
More Fiendish Bugs

● Same difficult class of bugs as failing to save context
  – Doesn't always occur
  – Doesn't always produce the same behavior
  – Hard to reproduce
  – Heisenberg Principle:
    • Addition of debugging code can alter behavior
Solution

- Disable interrupts before accessing shared data
- Reenable interrupts afterwards
- Compilers cannot reliably determine when such action is necessary.
  - Up to the programmer to thoroughly understand code and disable/reenable interrupts when appropriate.
Atomic code

- Disabling and renabling interrupts produces an “atomic” section of code.
  - Atomic: not being able to be divided (interrupted).
- Large atomic section affect Interrupt Latency
Goals

• Interrupt routines should remain as small as possible.
  • Every instruction adds time to process
• Interrupt routines should need to save/restore as little context as possible
  • Every register requires two instructions to save and restore
• Atomic sections should also remain small
  • To minimize delays in responding to interrupts