Syllabus

- Introduction
- Wiki Definition
- Interrupt Definition
- Reorder Buffer
- Priorities
- Reentrant & Recursive
- Bibliography
Introduction

- During execution of a piece of software, a number of unexpected events can happen in computing environment or on CPU
- A user may enter a character on a keyboard that is to be read
- Electric power, supplying the computer system, may suddenly disappear
- Given time slice granted to process p in time-share system may run its course; then p is suspended temporarily; may need future processor time slices!
- The CPU intends to execute the target of a branch at instruction pc, yet that code at pc is not resident
Introduction

- During a **floating-point divide**, the denominator is found to be 0.0, causing a result that is not representable, not computable: causing an **error**
- Two numbers are multiplied, yet the mathematical **product exceeds the numeric capacity** of the machine to store that value; user wishes to know!
- Elements of an array are **added** up, yet the next addition exceeds the numeric range of the machine
- The target address of some branch instruction lies **outside the code range** of the current program
- The object code for an x86 CPU executes an **INT instruction**, asking for OS support, parameterized via a defined machine register
Introduction

- Numerous causes, why a *regular flow* of execution changes; *regular execution* means SW executed by HW, as if no unusual event ever had happened

- Some events are unpredictable; such as power outage; they cause *interrupts*, are addressed by an interrupt handler, AKA interrupt service routine (ISR)

- Other events can be anticipated, yet not be known a-priori to happen; e.g. overflow of numeric value during floating-point add: Causes an *exception*; such exceptions can and must be handled; these are *not the same as interrupts*

- Yet other events are purposely programmed; e.g. OS request for some *system function*. Those are completely predictable. For example, the *INT call* on x86, or the request to read a character from an input device. Despite their name (such as INT ☺ on some systems), these actions are *not INTerrupts*
Introduction

Responding to Exceptional conditions: via interrupt, exception, or a system call; picture from Wikipedia
Interrupt

- **Definition:** Interrupt is a transparent, non-scheduled change in execution flow with a specific cause outside the interrupted program, treated by an interrupt handler, ending up at original program again.

- **Is unpredictable:** Programmer does not know that, when, why, where, how long interrupt happens.

- **Is unexpected:** Programmer does not know when interrupt happens, or whether it happens; it is asynchronous.

- **Cannot be pinpointed (i.e. coded) by location:** Programmer does not know where such an interrupt happens.

- **Cause** is only known once interrupt happens; is passed to ISR by HW, yet interrupted process does not know a-priori that it will happen; doesn't know afterwards it did happen.

- Cause can be external event like power-outage, time-slice consumption (timer interrupt), or other.
Interrupt Requirements

- After handling the interrupt, regular SW execution continues at the place of code right after the interrupted code location – sounds straight-forward?

- **Challenge**: if interrupt happens during execution of very long instruction, say the move of a large portion of memory via **byte-move** instruction!

- Challenge is handling the interrupt swiftly, perhaps *faster* than the time needed for the execution of some complex (long lasting) machine instruction!

- Interrupt must be handled so that the program never “knows” it was interrupted: i.e. the event must be completely **transparent**

- Except for the longer time to completion
Interrupt Requirements

- Responding to **interrupt must be fast**: cause may be catastrophic event, such as power outrage! In such case data must be saved, the complete program status must be saved, so processor may resume execution at a later time.

- At completion, it is not known that an interrupt has happened, except somehow execution ends up **more slowly** than expected; slower than if the interrupt had not happened; final result must be the same!

- **Summary**: the SW does not see **that, why, when, how often, how long, where in the code** an interrupt has happened; it is **not a programmed event**.

- **Note**: x86 INT instruction is not an interrupt! Though it is named a “**software interrupt**” instruction; is it totally predictable, locatable!

- **Note**: **System calls and exceptions are not interrupts!**
Interrupt: Wiki Definition

- “In system programming, an interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. An interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing. The processor responds by suspending its current activities, saving its state, and executing a function, called an interrupt handler (or an interrupt service routine, ISR) to deal with the event. This interruption is temporary, and, after the interrupt handler finishes, the processor resumes normal activities. There are two types of interrupts: hardware interrupts and software interrupts.”
Interrupt: Formal Definition

• A system **interrupt** is an asynchronous change of execution flow, caused by an external event; external to the currently running process!

• Interrupted SW completes execution transparently, as if the event never had happened, except for lost time

• Dedicated special system SW, named the **interrupt handler** (AKA interrupt service routine **ISR**), responds to interrupt

• We say, the ISR “handles” the interrupt
Exceptions, such as a floating-point overflow, are similar to interrupts, yet are caused by specific instructions, not by external events.

System calls are completely predictable, like other calls, and have little in common with interrupts or exceptions.

Again: the x86 INT instruction is no interrupt!
Reorder Buffer

- Reorder Buff is CPU resource that processes micro-operations in any order, but retires them in original program order.

- On a pipelined architecture there exist at any moment numerous instructions $pc_i$, $i = 1..n$ in various stages of partial completion; $n$ being the pipeline depth.

- When interrupt happens at instruction $pc_i$, all instructions before $pc_i$ must complete, and none of the instructions after $pc_i$ finish, til interrupt is handled.

- For any exception, its cause is attributable to a specific instruction.
Reorder Buffer

- When a genuine interrupt happens on a pipelined architecture, HW must decide, at which of the various $pc_i$ instructions the cut-off is to be made.
- So all operations before $pc_i$ can complete, and all started instructions after $pc_i$ get flushed to be completely resumed later.
- Note that Alpha μP by default (!) handles interrupts imprecisely! Good for performance!
- Contemporary architectures are practically all pipelined, even super-pipelined.
- Making correct interrupt handling hard!
Reorder Buffer

When an interrupt happens at time 6, did $i_1$ cause it, or was it $i_2$? And what should happen with $i_3..i_6$?

Horizontal: time, units of cycles. Vertical: consecutive instructions

<table>
<thead>
<tr>
<th>Instruct.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>i8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if</td>
<td>de</td>
<td>op1</td>
<td>op2</td>
<td>exec</td>
<td></td>
<td>wb</td>
<td></td>
</tr>
<tr>
<td>i7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if</td>
<td>de</td>
<td>op1</td>
<td>op2</td>
<td>exec</td>
<td></td>
<td>wb</td>
<td></td>
</tr>
<tr>
<td>i6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if</td>
<td>de</td>
<td>op1</td>
<td>op2</td>
<td>exec</td>
<td></td>
<td>wb</td>
<td></td>
</tr>
<tr>
<td>i5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if</td>
<td>de</td>
<td>op1</td>
<td>op2</td>
<td>exec</td>
<td></td>
<td>wb</td>
<td></td>
</tr>
<tr>
<td>i4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if</td>
<td>de</td>
<td>op1</td>
<td>op2</td>
<td>exec</td>
<td></td>
<td>wb</td>
<td></td>
</tr>
<tr>
<td>i3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if</td>
<td>de</td>
<td>op1</td>
<td>op2</td>
<td>exec</td>
<td></td>
<td>wb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If</td>
<td>de</td>
<td>op1</td>
<td>op2</td>
<td>exec</td>
<td></td>
<td>wb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>If</td>
<td>de</td>
<td>op1</td>
<td>op2</td>
<td>exec</td>
<td></td>
<td>wb</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6 CPI from here: 1 clock per new instruction retirement, CPI = 1
The μP manages a so-called reorder buffer to sort the *yet to be completed* vs. the *to be restarted* instructions.

Reordering requires sorting (by numbering) instructions in progress; in progress means: instructions started, but not yet retired.

Though instructions have been broken into smaller internal HW μ-ops, via numbering they can be reconstructed after handling the interrupt and ordered in original position to satisfy *sequential order* requirement.
Reservation Stations and Reorder Buffer

From Shen and Lipasti
Priorities

- When an **exception** occurs, such as a zero-divide, the time to respond is uncritical; i.e. waiting a few microseconds longer, even milliseconds, generally causes no new or real problem.

- Some exceptions are **ignored**, e.g. integer overflow on some Unix operating systems.

- When an **interrupt** occurs, the handler must decide, how and with which priority to react!

- For example, new data may have arrived at some input port, are available for a very limited, defined duration, to be consumed by HW or else being lost.

- **Loss being fatal** for that piece of information!
Priorities

- Moreover, an interrupt may happen while some other interrupt is being handled.
- Therefore, interrupts need priorities to sort out, which action to perform first, and which other actions can be postponed.
- To limit the time of postponement, interrupt handling must always be swift.
- Some interrupts are so critical (carry such high priority) that their handling cannot tolerate further interruptions.
- Hence it must be possible on a system to disable further interrupts.
- After handing, interrupts can again be enabled.
Disable Interrupts

- ISRs do **NOT** interrupt themselves, hence ISR code doesn’t need to be **recursive**
- Recursive would be OK, reentrant is sufficient
- When ISR is active, **nested** interrupts generally are possible
- Some interrupts are of such high priority, that further **interrupts** should **no longer** be tolerated, even when events triggering them do happen
- In such cases, interrupts are “disabled”; note: a brute-force HW action!
- Since the ISR is not a piece of user code, but code running in **supervisor mode**, such “interrupt disabling” is allowed, even **mandatory**!
Interrupt Handler (ISR)

- Causes for **interrupts** are plentiful
- How does interrupt service routine (ISR) decide, which action to take?
- Each type of interrupt is assigned its own **ID** number; ID used as index into table of service routines; more accurately, a table of addresses of service routines: AKA branch table
- Thus the data structure for selecting the ISR is uniform, the handling is fast, costing one added branch in addition to saving the machine state
Interrupt Handler (ISR)

Table of addresses of individual handlers for a specific processor: **ARM**

<table>
<thead>
<tr>
<th>Exception</th>
<th>Mode</th>
<th>Vector Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>SVC</td>
<td>0x00000000</td>
</tr>
<tr>
<td>Undefined Instruction</td>
<td>UND</td>
<td>0x00000004</td>
</tr>
<tr>
<td>Software Interrupt</td>
<td>SVC</td>
<td>0x00000008</td>
</tr>
<tr>
<td>Abort (Instruction Fetch Fault)</td>
<td>ABT</td>
<td>0x0000000C</td>
</tr>
<tr>
<td>Abort (Data Fetch Fault)</td>
<td>ABT</td>
<td>0x00000010</td>
</tr>
<tr>
<td>Vector Reserved</td>
<td></td>
<td>0x00000014</td>
</tr>
<tr>
<td>IRQ Interrupt</td>
<td>IRQ</td>
<td>0x00000018</td>
</tr>
<tr>
<td>FIQ Interrupt</td>
<td>FIQ</td>
<td>0x0000001C</td>
</tr>
</tbody>
</table>
Interrupt on ARM

- **ARM:** Advanced RISC Machine; before that: Acorn RISC Machine
- FIQ specific to ARM architecture, supporting two types of interrupts:
  - FIQs for fast, low latency interrupt handling
  - Interrupt Requests (**IRQs**), for more general interrupts
- FIQ takes priority over an IRQ on ARM CPU
- Generally ISRs differ from one manufacturer to others
Interrupt Handler (ISR)

- ISR requires global state (bit) to decide and communicate: whether interrupts are enabled

- ISR needs to know interrupt cause, such as:
  - Timer expiration
  - I/O waiting
  - Power loss, etc.

- ISR requirement: save state, be short to handle, handle transparently, return to place of interrupt after handing, as if nothing had happened, restore original state

- After completion, interrupt enable bit must unconditionally be reset to: on
Return From Interrupt

- Similar to the call-return mechanism, a completed ISR must **continue execution** – after handling the interrupt cause – at instruction after the recently *interrupted*, yet by now *completed* one

- Since interrupt has to be **transparent**, complete machine state is recreated at the point of return from saved information; similar to call-return

- Hence more than just the return address is saved by the ISR; the whole **processor state is saved**, to be restored after handling

- On some processors (e.g. **ARM**) the ISR has its own stack, different from the program stack
Reentrant & Recursive

- Code of an **ISR** does not need to be recursive
- As there is no need for the interrupt handler to call itself, directly or indirectly
- However, the ISR must be written in a way that it may be **called before having completed** the current interrupt!
- Yet the flow of control is that of a single process, not multiple parallel processes
- Interrupting the ISR must be allowed, since some sudden **higher priority interrupt** may have to be handled before the current one completes
- This dictates at least **reentrant, single-threaded code**; recursive code would be ok
Reentrant code may **not communicate via globals**; ISR will save and restore program state; certain high priority interrupts may **disable further interrupts**; handling must be fast; to share data: use **semaphores** for exclusivity

![Diagram showing reentrant code execution with interrupt handling]

- User code
- ISR 1
- int. 1
- User code
- ISR 1
- int. 2
- ISR 1
- int. 3
- ISR 2
- ISR 3
- Disable interrupts
- Enable interrupts & return & restore
- Return & restore
Reentrant & Recursive

- Straight-forward to design recursive source code in a high-level language
- Recursion is enabled by the type of object code generated; generally the compiler’s responsibility
- Recursion implemented by use of a run-time stack, with each incarnation of a recursive function having its own activation record, AKA stack frame (own locals, own formals, own return address and dynamic link)
- Reentrant code must adhere to coding restrictions
  - E.g. not to communicate via global objects
  - But reentrant code does not call itself, either directly or indirectly
Bibliography

3. ARM web site: https://www.arm.com/products/processors
Appendix: Definitions
Definitions

Exception

- An exception $e$ is an event caused by an irregular or faulty operation of a running program $p$. It is generally $p$ itself that causes $e$.

- Similar to an interrupt, $e$ causes the HW — or special system SW — to respond, to correct the situation where possible, or else to report the cause, location, and circumstance of $e$.

- Some exceptions are ignored, for example integer overflow on Unix systems, but the resulting arithmetic results generally are wrong.

- Location and reason for an $e$ are clearly identifiable by the instruction that caused it.
Definitions

Interrupt

- An interrupt is a transparent change in the normal flow of SW operation.
- SW does not know, when, that, why, where, how often or how long such an interrupt happens.
- The cause of the interrupt is outside the interrupted SW.
- Special piece of system SW called interrupt service routine (ISR) handles the interrupt and ensures that execution continues normally, once the interrupt has completed.
- During next execution with identical data, this interrupt might not happen at all.
Definitions

Machine-Check Error

- Machine-Check Error (MCE): a catastrophic event, detectable by the HW that causes system to end its regular path of execution.

- For example, a processor may get too hot, or the clock for a specific CPU may run too fast (creating heat) for continued, reliable operation.

- Result of an MCE is usually an end to further operation; a good MCE handler provides information about cause and location, before the machine shuts down.
Definitions

Recursive Code

- A algorithm – and thus a piece of SW – is recursive, if it is partly defined in simpler versions of itself.
- This implies that SW may call itself directly or indirectly, yet continues to make progress.
- For SW to be recursive it also has to be reentrant.
Definitions

Reentrant Code

- Reentrant SW can be called again before completing the current execution path
- To write reentrant code, communication via globals should be excluded; else sharing global information requires protection via semaphores
- Wiki: “a procedure is reentrant if it can be interrupted in the middle of its execution and safely be called again ("re-entered") before its previous invocations complete. The interruption could be caused by an internal action such as a jump or call, or by an external actions. Once the reentered invocation completes, the previous invocations will resume correct execution.”
Definitions

System Call

- A system call is a programmed (visible in the source program) request for system services
- The location is visible, as the service requires an explicit request via a programmed instruction
- For example, on the x86 architecture, the INT instruction is such a special request for service, though it is named an interrupt instruction!