# Microprocessor Microarchitecture
# Interrupt and Precise Exception

*Lynn Choi*

*Dept. Of Computer and Electronics Engineering*

# Interrupts

- ***Interrupts***
  - Forced transfer of control to a procedure (*handler*) due to external events (*interrupts*) or due to an erroneous condition (*exceptions*)
  - Exception
    - Generated by the currently running process due to an erroneous condition
    - Synchronous, internal
  - Interrupt
    - Asynchronous, external events
- ***Interrupt handling mechanism***
  - Allows interrupts/exceptions to be handled transparently to the executing process (application programs and operating system)
  - Procedure
    - When an interrupt is received or an exception condition detection, the current task is suspended and transfer automatically goes to a handler
    - After the handler is complete, the interrupted task resumes without loss of continuity, unless recovery is not possible or the interrupt causes the currently running task to be terminated.

# Exceptions

- ***Exception Classification (processor-generated)***
  - Fault
    - *Return to the faulting instruction*
    - Reported during the execution of the faulting instruction
    - Virtual memory faults
      - TLB miss, page fault, protection
    - Illegal operations
      - divide by zero, invalid opcode, misaligned reference
  - Trap
    - *Return to the next instruction* (after the trapping instruction)
      - For a JMP instruction, the next instruction should point to the target of the JMP instruction
    - Reported immediately following the execution of the trapping instruction
    - Examples: breakpoint, debug

# Exceptions

- Abort
  - Suspend the process at an unpredictable location
    - Does not report the precise location of the instruction causing the exception
    - Does not allow restart of the program
  - Severe errors or malfunctions
  - *Abort handlers* are designed to *collect diagnostic* information about the processor’s state and then perform a *graceful system shutdown*
  - Examples: bit error (parity error), inconsistent or illegal values in system tables

## ***Software-generated exception***

- *INT n* instruction generates an exception with an exception number (n) as an operand

# Precise Exception

- ***All exceptions except aborts must report the exception on a precise instruction boundary***
- ***Precise exception model***
  - All integer/FP exceptions are reported on the faulting instruction
  - All previous instructions are completed before the interruption point
  - All subsequent operations are nullified
  - After handling the exception, the execution resumes at the faulting instruction (fault) or at the next instruction (trap)
  - For O-O-O processors,
    - Interrupts are taken at the retirement phase of instruction execution; so they are always taken in-order.

# Exception Handling

- ***Exception (interrupt) vector***
  - Each exception or an interrupt is associated with an identification number, *vector*
- ***Exception procedure***
  - Flush all the instructions fetched subsequent to the instruction causing the exception condition from the pipeline
  - Drain the pipeline: complete all previous instructions
    - Complete all outstanding write operations prior to the faulting instruction
  - Save the PC of the next instruction to execute
  - Also need to save the necessary registers and stack pointers to allow it to restore itself to its state
  - Vector the interrupt
  - Fetch instruction from the ISR and service the interrupt
  - Return from the interrupt

# (External) Interrupt

- ***Interrupt***
  - Asynchronous
  - Caused by external events, IO devices
  - Return to the next instruction for a restart
- ***Interrupt Classification***
  - Maskable interrupt
    - Can be disabled/enabled by an instruction
    - Generated by asserting INTR pin or sending interrupt messages over the *APIC (Advanced Programmable Interrupt Controller)* bus
    - External interrupt controllers
      - Intel 8259 PIC (programmable interrupt controller) delivers the interrupt vectors on the system bus during interrupt acknowledge cycle

# Interrupt

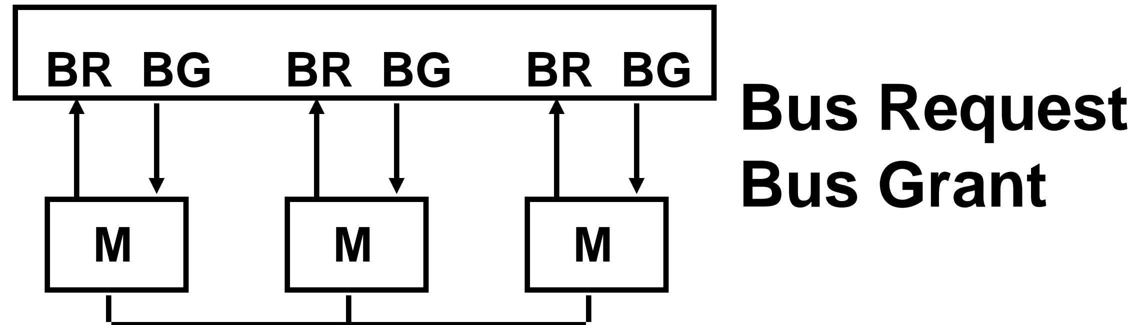
- Non-maskable interrupt (NMI)
  - Cannot be disabled by program
  - Received on the processor's NMI# input pin
- Software interrupt
  - Generated by INT n instruction
    - INT instruction can be used to generate an interrupt or an exception by using a vector number as an operand
  - Viewed as an implicit call to interrupt handler of interrupt vector n
  - No mechanism for masking interrupts

# Interrupt Priority

- ***Predefined order of different interrupts***
  - H/W Reset, Machine Check Abort
  - External HW interventions
    - INIT - like H/W reset without flushing caches)
    - SMI (System (e.g. power) Management Interrupt)
  - Traps on the previous instruction
  - External Interrupts - NMI, MI
  - Faults on executing an instruction
    - DTLB faults
    - FP exception, overflow, alignment
  - Faults from fetching/decoding an instruction
    - ITLB faults: page miss, access/protection violation
    - Illegal opcode
- ***Lower priority exceptions are regenerated after returning from the higher priority interrupt handler***

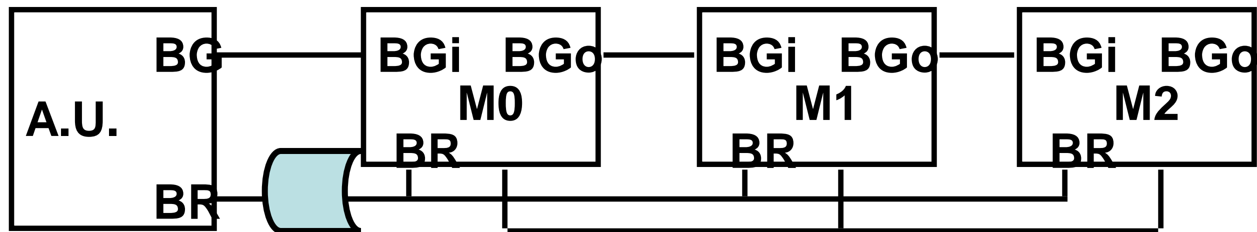
# Interrupt Priority

Parallel (Centralized) Arbitration (can use Priority Encoder)

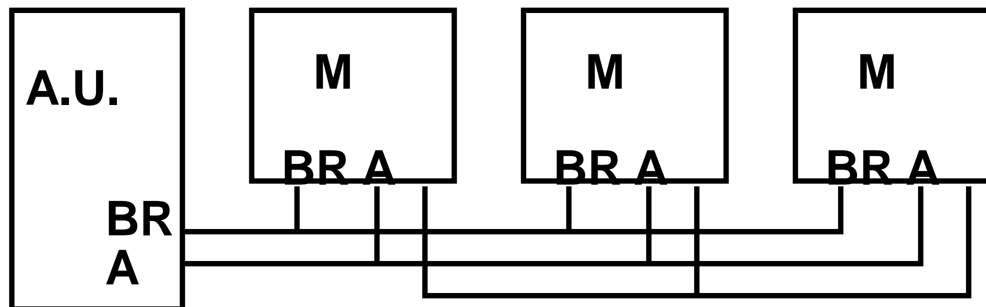



**Bus Request**
**Bus Grant**

Serial Arbitration (*Daisy Chaining*, M0 has the highest priority)




Polling
(by S/W)




*From UCB Patterson*

# Precise Interrupt

- ***Definition: precise interrupt***
  - An interrupt is *precise* if the saved processor state corresponds to a sequential model of program execution before the interrupt occurs
- ***Precise exception model***
  - All previous instructions are completed before the interruption point
  - All subsequent operations are nullified
  - After handling the exception, the execution resumes at the faulting instruction (fault) or at the next instruction (trap)
- ***Precise interrupt is difficult to implement for***
  - Pipelined processors
    - Because exceptions can be generated out of order in different pipeline stages
    - For both in-order and OOO processors
  - Out-of-order processors
    - Because instructions may complete before an instruction issued earlier raises an exception

# Precise Interrupt

- ***Precise interrupts are necessary to restart program execution after the unexpected event***
  - I/O and timer interrupts
  - virtual memory related interrupts, i.e. page faults, TLB miss, etc.
  - software debugging
  - graceful recovery from arithmetic exceptions, I.e. underflow, overflow

# Solutions for Precise Exceptions

- ***In-order pipelines***
  - All instructions pass through pipeline in program order and *exception conditions are reported in program order* at the end of pipeline before the processor state is modified
    - Instructions modify the processor state only when all previously issued instructions are known to be free of exceptions
- ***Out-of-order pipelines***
  - Early O-O-O machines such as CDC6600 and IBM 360/91 did not support precise exceptions in favor of maximum parallelism
  - Some machines such as early MIPS processor are *restartable* but did not support full precise interrupts. This requires implementation-dependent software shift through the machine dependent state and restore the pipeline state.

# Exception Recovery & Restart

- ***Motivation -***
  - *Precise interrupts* on *exception*
    - provide in-order state to the exception handler
- ***Exception recovery***
  - Cancel the effects of instructions that should have not been issued
  - Need to buffer states to restore the unspeculated states
  - Some exceptions are easy to handle since the exception conditions are detected prior to execution
    - Privilege interrupts, ITLB miss, etc.
    - Therefore, these exceptions should not report the exceptions until the end of pipeline
  - Exception conditions may be detected out-of-order!
    - But exceptions must be reported in program order

# HW Schemes

- **Checkpoint repair - *Hwu & Patt***
- **Reorder buffer - *Smith & Pleszkun***
  - Instructions complete out-of-order but commit in order
- ***Variations of Reorder buffer***
  - *History buffer* - Smith & Pleszkun
  - *Reorder buffer with Future file* - Smith & Pleszkun

# Buffering States

- ***In-order state***
  - The most recent assignments performed by the longest continuous sequence of completed instructions
  - Redundant assignments are superseded
  - Necessary for in-order completion
- ***Lookahead state***
  - The first uncompleted instruction to the end of the instruction sequence including all pending assignments
  - No value is superseded
- ***Architectural state***
  - The most recently completed and pending assignments to each register
  - State used by a following instruction
  - Can be obtained by combining in-order & lookahead states

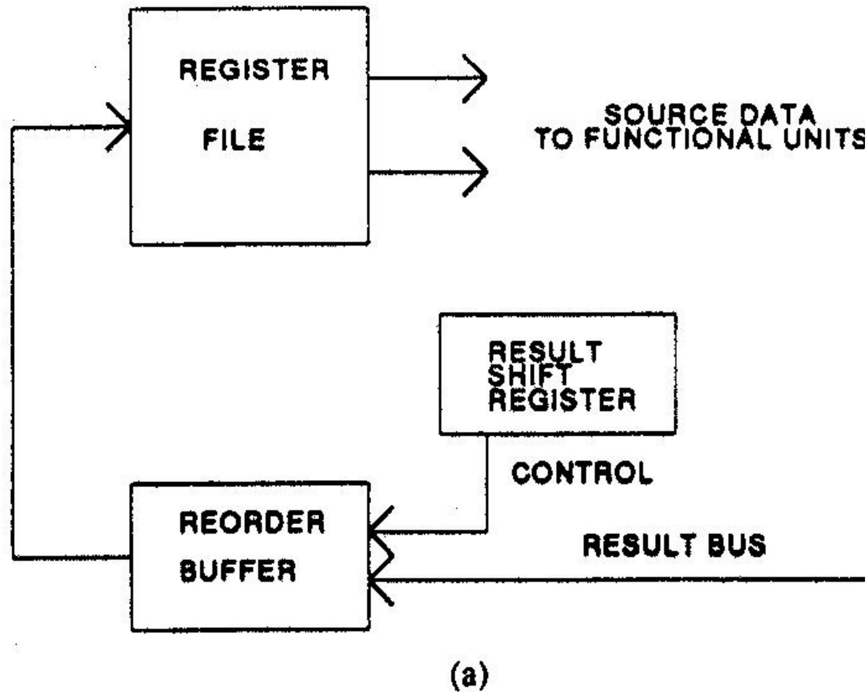
# HW Schemes

- ***Checkpoint repair***
  - current logical space - architectural state
  - backup space - in-order state
- ***Reorder buffer***
  - Register file - in-order state
  - Reorder buffer - lookahead state
- ***History buffer***
  - Register file - architectural state
  - History buffer - in-order state
- ***Reorder buffer with Future file***
  - Register file - in-order state
  - Reorder buffer - lookahead state
  - Future file - architectural state

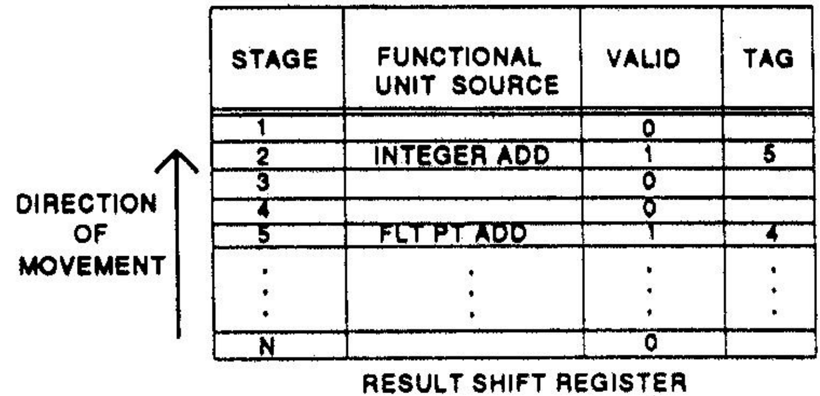
# Reorder Buffer - Smith & Pleszkun

- ***Maintain two states***
  - Register file contains the in-order state
  - Reorder buffer contains the lookahead state
    - The architectural state is obtained by combining in-order and lookahead states.
    - Multiple assignments can exist for a single register
    - Managed as FIFO queue
      - When an instruction is decoded, an entry is allocated on the top of the reorder buffer. After the instruction completes, the result is written to the entry.
        - Instruction decoding stalls if there is no reorder buffer entry
      - When the value reaches the bottom, it is written into the register file if there is no exception. If the instruction is not complete, the reorder buffer does not advance until the instruction completes.
      - If there is an exception, the reorder buffer contents are discarded and then reverts to the in-order state in the register file.
    - Each entry contains its PC to provide PC on exception
  - On an exception
    - Locate an exception point in the reorder buffer to know which entries to reset
  - On a branch
      - Allocate a reorder buffer entry for each branch, even though the branch does not produce result
    - On a branch misprediction
      - Should not discard the entire reorder buffer; some of the lookahead states are for instructions preceding the branch.

# Reorder buffer



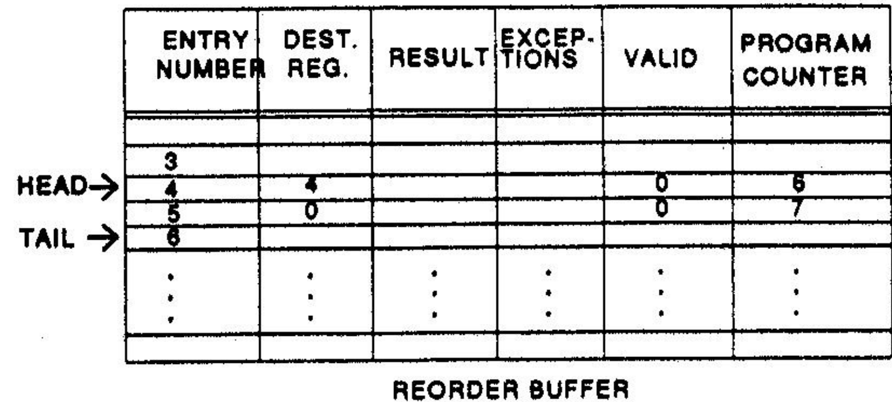

(a)



DIRECTION OF MOVEMENT

| STAGE | FUNCTIONAL UNIT SOURCE | VALID | TAG |
|---|---|---|---|
| 1 | | 0 | |
| 2 | INTEGER ADD | 1 | 5 |
| 3 | | 0 | |
| 4 | | 0 | |
| 5 | FLT PT ADD | 1 | 4 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| N | | 0 | |

RESULT SHIFT REGISTER



| | ENTRY NUMBER | DEST. REG. | RESULT | EXCEP-TIONS | VALID | PROGRAM COUNTER |
|---|---|---|---|---|---|---|
| | | | | | | |
| | 3 | | | | | |
| HEAD→ | 4 | 4 | | | 0 | 6 |
| | 5 | 0 | | | 0 | 7 |
| TAIL → | 6 | | | | | |
| | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| | | | | | | |

REORDER BUFFER

(b)

*IEEE All rights reserved*

# Reorder Buffer

- ***Disadvantages***
  - Require associative lookup to combine in-order and lookahead states
    - It is possible for more than one entry in the reorder buffer to correspond to source register
      - Only the *latest* reorder buffer entry need to be bypassed
    - Can be solved by an additional buffer called *future file*
  - The instruction dispatch logic does not know the original instruction order, and so the dispatch logic cannot prioritize instructions for issue based on this order.
- ***Advantages***
  - No need to keep instructions ordered in the central window
    - The window need not be compressed as instructions are issued. Instead, new instructions from the decoder are simply placed into the freed locations.
- ***Reorder buffer used in***
  - HP PA-8000
  - DEC Alpha 21264
  - Intel P-Pro and Pentium II, III
  - MIPS R10000
  - Power PC620

# Exercises and Discussion

- ***Compare the following: TLB miss, PTE miss, page miss***
- ***Assume that you have a Pentium 4 processor, which has a 3-way superscalar 20-stage OOO pipeline. At a particular cycle, there are 1 exception (iTLB miss), 1 external interrupt from DMA, and 1 branch misprediction at the same time. How does the Pentium 4 handle these 3 events properly?***