Dynamic Scheduling

- Static scheduling
  - Schedule instructions at compiler time to get the best execution time

- Dynamic scheduling
  - Hardware changes instruction execution sequence in order to minimize execution time (out of order execution)
  -Dependences must be honored
  - For the best result, false dependences must be removed
  - For the best result, control dependences must be tackled

- Components of dynamic scheduling
  - Check for dependences ⇒ “do we have ready instructions?”
  - Select ready instructions and map them to multiple function units

- Instruction window
  - When we look for parallel instructions, we want to consider many instructions (in “instruction window”) for the best result
  - Branches hinder forming a large, accurate window
FIGURE 4.75 The A8 pipeline. The first three stages fetch instructions into a 12-entry instruction fetch buffer. The Address Generation Unit (AGU) uses a Branch Target Buffer (BTB), Global History Buffer (GHB), and a Return Stack (RS) to predict branches to try to keep the fetch queue full. Instruction decode is five stages and instruction execution is six stages.
The Intel Core i7 architecture

FIGURE 4.77 The Core i7 pipeline with memory components. The total pipeline depth is 14 stages, with branch mispredictions costing 17 clock cycles. This design can buffer 48 loads and 32 stores. The six independent units can begin execution of a ready RISC operation each clock cycle.

Handling exceptions (Section 4.9)

- Exceptions/interrupts are an important part of a processor
- Modern OS and debuggers depend on the hardware support for exceptions
- Interrupt mechanism allows efficient use of CPU cycles by allowing the CPU to do useful work while waiting for an event.

  - Precise exception
    - Execute all instructions before the faulting instruction
    - Do not execute the faulting instruction and the following ones
    - Resume from the faulting instruction after the exception is taken care of

- How does MIPS handle exceptions?
  - Record the PC of the next instruction in a special place (the EPC register)
  - Record the cause of the exception (the Cause register)
  - Squash the instructions (inclusive of the faulting one)
  - Jump to a pre-determined handling routine (PC = 8000 00180)
  - When done, the handling routine will execute a “return from interrupt” to resume from the instruction following the faulting instruction (EPC → PC), or the faulting instruction (EPC-4 → PC) – depending on implementation.
The Handler

- Typically, the exception handler will
  - Determine action required
  - If not fatal
    - Take corrective action
    - use EPC to return to program
  - Otherwise
    - Report error using EPC, cause, ...
    - Terminate program
- Handler jumps to an address which is determined by the cause
  - Addresses are stored in a vector which is indexed by the cause
  - Example:

```
Undef opcode: C000 0000
Overflow: C000 0020
...: C000 0040
```

For external interrupts: the handler address is stored in a vector which is indexed by the interrupt type. The PC+4 is saved in EPC and the handler’s address replaces the current PC.

Exceptions in a Pipeline

- Another form of control hazard

- Consider overflow on add in EX stage
  
  \[
  \text{add } 3, 2, 1
  \]
  
  - Prevent 3 from being clobbered
  - Complete the instructions that entered the pipeline before the "add"
  - Flush the "add" and subsequent instructions
  - Set the Cause register
  - Save PC+4 in EPC register
  - Transfer control to handler

- Similar to miss-predicted branch
  - Use much of the same hardware
What is instruction level parallelism (ILP)?

- Execute *independent instructions* in parallel
  - Provide more hardware function units (e.g., adders, cache ports)
  - Detect instructions that can be executed in parallel (in hardware or software)
  - Schedule instructions to multiple function units (in hardware or software)
- Goal is to improve instruction throughput
- Pipelining (a single pipeline) is a form of ILP which ideally gives CPI = 1
- With multiple pipelines, we can achieve CPI < 1 (IPC > 1)

- How does it differ from task-level parallelism (discussed in section 6)?