Lecture 5: Out-of-Order Processing

Lisa Wu, Krste Asanovic

http://inst.eecs.berkeley.edu/~cs252/sp17
Last Time in Lecture 4

- Iron Law of processor performance
- Pipelining: reduce cycle time, try to keep CPI low
- Hazards:
  - Structural hazards: interlock or more hardware
  - Data hazards: interlocks, bypass, speculate
  - Control hazards: interlock, speculate
- Precise traps/interrupts for in-order pipeline
Review: The Definition of Precise Exceptions/Interrupts

- The architecture state needs to be consistent when the exception traps or interrupts need to be processed
  - All instructions preceding the instruction pointed to by the EPC (saved PC) need to have been executed and architectural states updated
  - All instructions following the instruction pointed to by the EPC need to have not been executed and have not modified any architectural states
Review: Why is Precise Exception/Interrupt Handling Important?

- Enable restartable processes upon an external interrupt
- Enable recovery/restarting following an exception such as a page fault in a system with virtual memory
- Help in isolating the exact exception condition in software debugging
- Enable traps into software handlers: e.g. simulation of unimplemented opcodes, graceful recovery of arithmetic (floating point) exceptions per IEEE floating point standard
Review: Handle Exceptions in the Presence of Multi-Cycle Functional Units

- What happens if FMUL takes an exception?

Example by Onur Mutlu, CMU, Fall 2011
Review: Handle Exceptions in the Presence of Multi-Cycle Functional Units

- Possible solution: Make all functional units take the same amount of time
- Enabling precise exceptions and preserving sequential architectural model
- What about memory accesses?

Example by Onur Mutlu, CMU, Fall 2011
Solutions Presented in Smith and Pleszkun 1988

• In-order completion
  • Result shift register: solve the problem of structural hazard caused by having one result bus (one reg file write port)
  • Interlock at issue
  • What about stores (memory is part of the process state)

• Out-of-order completion
  • Reorder Buffer (ROB)
  • History Buffer
  • Future File
Reorder Buffer (ROB)

• Allow instructions to complete out-of-order, but commit in-order
• Circular buffer w/ head and tail pointers. Instructions between head and tail are valid.
• Allocate a new entry at the tail pointer when an instruction is issued
• When the issued instruction completes, the result(s) and the exception(s) are sent to the ROB
• The oldest instruction in the ROB (at the head) with no exceptions can commit, i.e. write to architectural states including register file and memory
Reorder Buffer (ROB)

- ROB still suffers a performance penalty: completed results are held in the ROB but can’t be used by other instructions until written back to the RF

Example by Onur Mutlu, CMU, Fall 2011
Bypass from ROB to functional units improve performance

Bypass logic needs CAMs to find ROB entry; too expensive
History Buffer

- Allow instructions to modify register file as results are completed out-of-order
- But retain enough information in a History Buffer so the architectural states can be restored when an exception needs to be handled
- Allocate a new entry at the tail pointer when an instruction is issued
- Destination register value needs to be read out prior to instruction execution and written into the History Buffer (needs an extra RF read port)
- The oldest instruction in the HB (at the head) with no exceptions can discard HB entry, otherwise write back old RF values from tail to head
• Good: HB only needs to be accessed on an exception, no cumbersome bypass logic as in the case of the ROB
• Bad: Needs an extra RF read port, rollback sometimes take a long time
Future File

- Keep two register files: one architectural RF (to be used on exception handling), one speculative RF (updated as soon as instructions complete)
- Still need the ROB for instruction in-order commits
- No rollback on exception, just copy the architectural RF content over to the saved states
Microarchitectural Design Considerations: Design patterns learned from handling precise exceptions/interrupts

- Resource contention
  - Reserving a hardware resource
- The sequential semantics of the ISA/architectural model need to be preserved
- Stores need to be handled with special care
- Keep a shadow copy as a speculative state
- Start with a functional, possibly low performing solution, and slowly move towards a functional and high performing solution
- Tradeoffs between performance, cost, and complexity
IBM 7030 “Stretch” (1954-1961)

- Original goal was to use new transistor technology to give 100x performance of tube-based IBM 704.
- Design based around 4 stages of “lookahead” pipelining
- More than just pipelining, a simple form of decoupled execution with indexing and branch operations performed speculatively ahead of data operations
- Also had a simple store buffer
- Very complex design for the time, difficult to explain to users performance of pipelined machine
- When finally delivered, was benchmarked at only 30x 704 and embarrassed IBM, causing withdrawal after initial deliveries
Concepts that Stretch Pioneered

• Memory Interleaving:
  • Address modulo 2 for the two instruction core memory (16K each)
  • Address modulo 4 for the four data core memory (16K each)

• Instruction and Data Prefetching:
  • ”Lookahead” for 2 instructions and 4 data operands (initiate these fetches from memory ahead of time)

• Precise Exception/Interrupt Handling
  • “Virtual Memory” and “Lookahead”: Prefetch buffer and store buffer, also like a combination of a history buffer and a set of instruction reservation stations. Implemented some form of load/store forwarding also.

• Predecoding:
  • Preliminary decoding of instruction classes to (pre-) execute certain class of instructions ➔ decoupled execution ➔ limited form of OoO execution

• Branch Prediction:
  • Speculatively execute conditional branch paths
Simple vector-vector add code example

# for(i=0; i<N; i++)
# A[i]=B[i]+C[i];

loop: fld f0, 0(x2) // x2 points to B
    fld f1, 0(x3) // x3 points to C
    fadd.d f2, f0, f1
    fsd f2, 0(x1) // x1 points to A
    add x1, 8 // Bump pointer
    add x2, 8 // Bump pointer
    add x3, 8 // Bump pointer
    bne x1, x4, loop // x4 holds end
Simple Pipeline Scheduling

Can reschedule code to try to reduce pipeline hazards

loop: fld f0, 0(x2) // x2 points to B
     fld f1, 0(x3) // x3 points to C
     add x3, 8 // Bump pointer
     add x2, 8 // Bump pointer
     fadd.d f2, f0, f1
     add x1, 8 // Bump pointer
     fsd f2, -8(x1) // x1 points to A
     bne x1, x4, loop // x4 holds end

Long latency loads and floating-point operations limit parallelism within a single loop iteration
Loop Unrolling

Can unroll to expose more parallelism

```assembly
loop:  fld  f0, 0(x2)  // x2 points to B
       fld  f1, 0(x3)  // x3 points to C
       fld  f10, 8(x2)
       fld  f11, 8(x3)
       add  x3, 16  // Bump pointer
       add  x2, 16  // Bump pointer
       fadd.d  f2, f0, f1
       fadd.d  f12, f10, f11
       add  x1, 16  // Bump pointer
       fsd  f2,  -16(x1)  // x1 points to A
       fsd  f12,  -8(x1)
       bne  x1, x4, loop  // x4 holds end
```

- Unrolling limited by number of architectural registers
- Unrolling increases instruction cache footprint
- More complex code generation for compiler, has to understand pointers
- Can also software pipeline, but has similar concerns
Decoupling *(lookahead, runahead)* in µarchitecture

Can separate control and memory address operations from data computations:

```
loop:  fld f0, 0(x2)  // x2 points to B
       fld f1, 0(x3)  // x3 points to C
       fadd.d f2, f0, f1
       fsd f2, 0(x1)  // x1 points to A
       add x1, 8     // Bump pointer
       add x2, 8     // Bump pointer
       add x3, 8     // Bump pointer
       bne x1, x4, loop  // x4 holds end
```

The control and address operations do not depend on the data computations, so can be computed early relative to the data computations, which can be delayed until later.
Simple Decoupled Machine

Integer Pipeline

Floating-Point Pipeline

{Load Data Writeback µOp}
{Compute µOp}
{Store Data Read µOp}

Check

Load Address

Store Address Queue

Load Data

Store Data Queue

µOp Queue

Load Data Queue

D X1 X2 X3 W
Decoupled Execution

- `fld f0` → Send load to memory, queue up write to f0
- `fld f1` → Send load to memory, queue up write to f1
- `fadd.d` → Queue up `fadd.d`
- `fsd f2` → Queue up store address, wait for store data
- `add x1` → Bump pointer
- `add x2` → Bump pointer
- `add x3` → Bump pointer
- `bne` → Take branch
- `fld f0` → Send load to memory, queue up write to f0
- `fld f1` → Send load to memory, queue up write to f1
- `fadd.d` → Queue up `fadd.d`
- `fsd f2` → Queue up store address, wait for store data

Check load address against queued pending store addresses

Many writes to f0 can be in queue at same time
Performance

• Measure performance of executing an individual operation

• Measure performance of executing a program using instruction mix
  • Instruction mix = relative frequency of instructions in a program
  • Weighted average instruction execution time

• Use industry standard benchmarks such as SPECCPU2006
Speedup = \frac{1}{(R_{\text{serial}} + R_{\text{parallel}}/n)}
Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)