Energy-Aware Computing

Lecture 4: OoO Processors
review

Outline

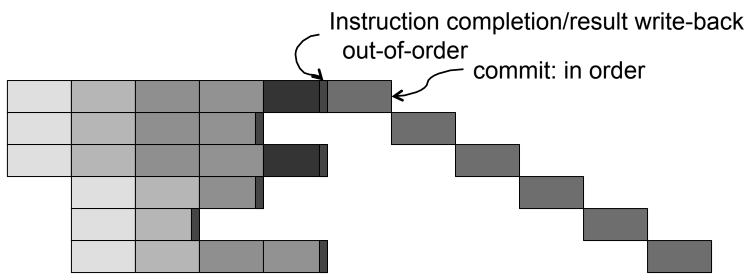
- OoO processor basics
- Hardware structures and principles
- Simplescalar modeling of OoO processors

Note: presentation tuned for simple-scalar
other solutions are possible

OoO overview

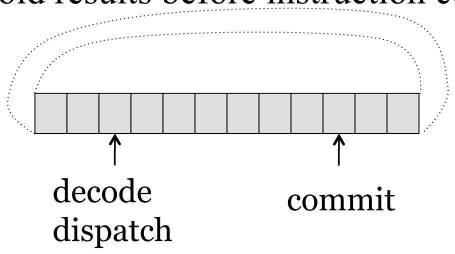
- *What*: execute many instructions simultaneously
  - More than overlapped execution (pipelining)
- *How*: multiple functional units (ALUs int, fp, memory ports, reg-file ports, wide fetch
- *Problems*: different execution latencies, instruction dependencies, precise exceptions, …

Different execution latencies



Re-order buffer (ROB)

- FIFO to track instructions in program order
  - Allocate entries at dispatch stage
  - De-allocate at commit
- Used for other purposes:
  - Hold results before instruction commits



Data dependencies

- True dependencies (RAW)
  - Must be preserved or program breaks
- Other dependencies (WAR, WAW)
  - Remove using *register renaming*
- Physical registers vs architectural registers
  - New physical register per instruction, in ROB
  - Rename hardware (RAT) maintains mapping

Register renaming

- At dispatch use RAT to:
  - look-up for current source operands mapping
  - write new mapping for destination register
- At commit
  - “remove” RAT entry
- Pipe squash (e.g. branch misprediction)
  - restore RAT to state before misprediction

Instruction dispatch stage

- Equivalent to decode stage
  - Instruction decoding
  - ROB allocation
  - Register renaming
  - Source operand reading (if available)
  - Dispatch to reservation station – essentially a queue of instructions waiting for operands and available FUs for execution

Instruction wake-up & selection

- Results from FUs get stored in ROB
- Dependent instructions monitor result busses and *wake-up*
  - become ready for execution
- Scheduler selects ready instructions and *issues* to functional units
  - *issue width* – max number of instr issued
  - *instruction window* – number of instr in the queue between dispatch and issue

Loads & stores

- Stores must change memory at commit
  - Why not earlier?
  - But too frequent to stall at each store
- Special Load/Store Queue
  - issue to LSQ and continue execution
  - when store @ commit, actually perform store
- Load/store dependencies
  - put loads in LSQ in program order
  - loads may forward store values
  - in multi-proc, mem-op ordering is crucial

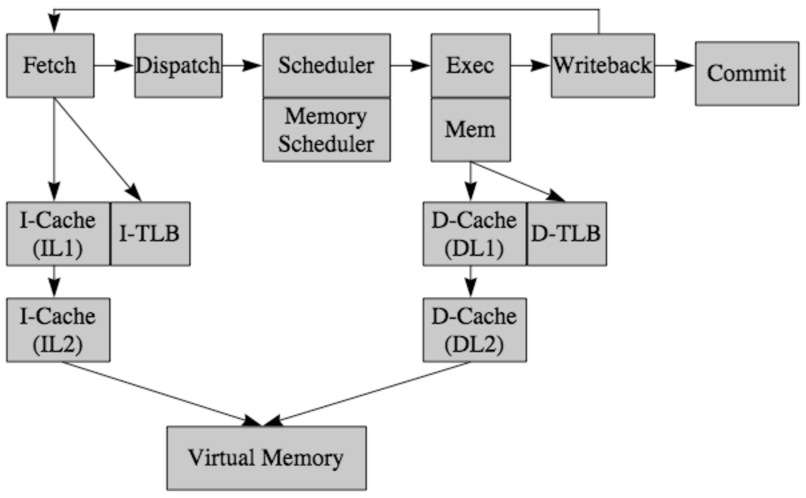
Branches and speculation

- Branch prediction, speculative execution to keep machine working at high speed
- Fetch asks bpred hardware for next PC
- When branch executes, checks the prediction
  - if wrong, give IFU correct PC, squash pipe
- bpred hardware needs updating
  - Three main options: @dec, @writeback, @commit

Branch prediction hardware

- RAS – return address stack
  - simple stack for providing return addresses
- BTB – branch target buffer
  - a cache proving target PC for known branches
- Branch direction prediction
  - lots of options. Simplescalar has built-in support for a number of them.

Simplescalar architecture



Simplescalar is a simulator

- Don’t forget: Simplescalar does not model hardware directly
  – it takes shortcuts to speed up simulation
- Instructions are actually “executed” early
  – in-order during dispatch (decode) stage
- Misspeculated instructions are updating a “fake” reg file and memory
- Caches only keep tags and status information
  – Actual data is kept in memory

Mapping into Simplescalar

- Reorder buffer (incl. physical registers), reservation stations are combined into a Register Update Unit (RUU)
- Loads/stores create 2 micro-ops
  - effective address calculation (could be reg+reg) in RUU
  - actual load/store in LSQ

RUU

- Circular buffer, each entry contains
  - The instruction opcode, PC
  - Ready bits for source registers
  - A linked list of consumers per destination register
  - Info for recovering from branch misprediction
  - Status flags, e.g., what state is this in, is it an address op
- An instruction can execute when all source registers are available: readyq in ruu_issue()
- On write-back:
  - walk target list, set ready bits of consumers

Renaming mechanism

- Register renaming combined with dependency lists
- CREATE_VECTOR(reg_name) returns current RUU entry which (will) produce the result
  - Note: it does not create an entry!