Superscalar- processing

Stallings 2010: Ch 14
- Instruction dependences
- Register-renaming
- Pentium / ARM

Superscalar processors

- Goal:
  - Concurrent execution of scalar instructions
- Several independent pipelines:
  - Not just more stages in one pipeline
  - Own functional units in each pipeline

Superscalar processor

- Efficient memory usage:
  - Fetch several instructions at once, prefetching
  - Data fetch and store (read and write)
  - Concurrency
- Several instructions of the same process executed concurrently on different pipelines:
  - Select executable instruction (ready for execution stage) from the prefetched instructions following a policy
  - Instructions may complete in different order than started
- Finish more than one instruction during each cycle
- When is it ok for an instruction finish before the preceding ones?

Effect of Dependencies

- True Data/Flow Dependency (datariippuvuus)
- Read after Write (RAW)
- Instruction after the jump executed only, when jump does not happen
- Superscalar pipeline has more instructions to waste
- Variable-length instructions: some additional parts known only during execution
- Resource Conflict (Resurssiriippuvuus)
  - One or more pipeline stage needs the same resource
  - Memory buffer, ALU, access to register file, ...
Effect of dependencies

WAR and WAW Dependencies

Dependencies Specific to Out-of-order Completion

How to Handle Dependencies?

ILP vs. Machine Parallelism

Superscalar execution
Superscalar Execution
- Instruction fetch (käskeyjen nouto)
  - Branch prediction (hyppyjen ennustus)
  - Dispatch to instruction window (valintatallennus)
- Instruction issue (käskyn päästämisen hiljalle)
  - Check (and remove) data, control and resource dependencies
  - Reorder; issue suitable instructions to pipelines
- Pipelines proceed *without waits*
- Instruction complete, retire (suoritus valmistuu)
  - Commit or abort (hyväksy tai hylkää) *usually all state changes occur here*
  - Check and remove write dependencies and antidependencies
  - Wait / reorder (järjestä uudelleen)
Solve Dependencies by Register Renaming

- Some dependencies are caused by register names, not data
  - The same name could be used for several independent elements
  - Thus, instructions have unintended write and antidependencies
  - Causing unnecessary waits

Solution: Register renaming

- Hardware must have more registers (than visible to the programmer and compiler)
- Hardware allocates new real registers during execution in order to avoid name-based dependencies

Need

- More internal registers (register files, register set), e.g. Pentium II has 40 working registers
- Hardware can allocate and manage registers, and perform the mapping dynamically at execution time

Register Renaming

Output dependency (WAW):
- i3 must not write R3 before i1 writes R3
  \[ R3 \leftarrow R3 + R5 \]  \[ R3 \leftarrow R3 + 1 \]

Anti dependency (WAR):
- i3 must not write R3 before i2 has read the value from R3
  \[ R3 \leftarrow R3 + R4 \]

Solution

- Rename R3
- Use work registers R3a, R3b, R3c
  - Other registers similarly: R4b, R5a, R7b
  - No more name dependencies!
Lecture 9: Superscalar Processing

**Generation of Pentium Pipeline µops**

a) Fetch IA-32 instruction from L2 cache and generate µops to L1
   - Uses Instruction Lookaside Buffer (i-TLB)
   - and Branch Target Buffer (BTB)
   - four-way set-associative cache, 512 lines
   - 1-µops (=118 bit RISC) per instruction (most cases),
     if more then stored to microcode ROM
b) Trace Cache Next Instruction Pointer - instruction selection
   - Dynamic branch prediction based on history (4-bit)
   - If no history available, Static branch prediction
     - backward, predict "taken"
     - forward, predict "not taken"
c) Fetch instruction from L1-level trace cache

d) Drive – wait (instruction from trace cache to rename/allocator)

e) Allocate resources
   - 3 micro-operations per cycle
   - Allocate an entry from Reorder Buffer (ROB) for the µops
     (128 entries available)
   - Allocate one of the 128 internal work registers for the result
   - And, possibly, one load (of 48) or store (of 24) buffer
   - Register renaming
     - Clear name dependencies by renaming
     - (16 architectural regs to 128 physical registers)
   - If no free resource, wait (+ out-of-order issue)
   - ROB-entry contains bookkeeping of the instruction progress
   - Micro-operation and the address of the original IA-32 instr.
   - State: scheduled, dispatched, completed, ready
   - Register Alias Table (RAT): which IA-32 register → which physical register

**Pentium Pipeline Resource allocation**

- Allocate resources
  - 3 micro-operations per cycle
  - Allocate an entry from Reorder Buffer (ROB) for the µops
    (128 entries available)
  - Allocate one of the 128 internal work registers for the result
  - And, possibly, one load (of 48) or store (of 24) buffer
  - Register renaming
    - Clear name dependencies by renaming
    - (16 architectural regs to 128 physical registers)
  - If no free resource, wait (+ out-of-order issue)
  - ROB-entry contains bookkeeping of the instruction progress
  - Micro-operation and the address of the original IA-32 instr.
  - State: scheduled, dispatched, completed, ready
  - Register Alias Table (RAT): which IA-32 register → which physical register

**Pentium Pipeline Window of Execution**

- Micro-Op Queueing
  - 2 FIFO queues for µops
    - One for memory operations (load, store)
    - One for everything else
  - No dependencies, proceed when room in scheduling
- Micro-Op Scheduling
  - Retrieve µops from queue and dispatch (issue) for execution
  - Only when operands ready (check from ROB-entry)
  - Dispatching (issue)
    - Check the first instructions of FIFO-queues (their ROB-entries)
    - If execution unit needed is free, dispatch to that unit
    - Two queues → out-of-order issue
    - max 6 micro-ops dispatched in one cycle
  - ALU and FPU can handle 2 per cycle
  - Load and store each can handle 1 per cycle

**Pentium Integer and FP Units**

- Get data from register or L1 cache
- Execute instruction, set flags (unlike)
- Several pipelined execution units
  - 4 * ALU, 2 * FPU, 2 * load/store
  - E.g. fast ALU for simple ops, own ALU for multiplications
  - Result storing: in-order complete
  - Update ROB, allow next instruction to the unit
- Branch check
  - What happened in the jump/branch instruction
  - Was the prediction correct?
  - Abort incorrect instruction from the pipeline (no result storing)
- Drive – update BTB with the branch result

**Pentium 4 Hyperthreading**

- One physical IA-32 CPU, but 2 logical CPUs
- Instructions from 2 processes in the same pipeline
- OS sees as 2 CPU SMP (symmetric multiprocessing)
  - Logical processors execute different processes or threads
  - No code-level issues
  - OS must be capable to handle more processors
    (like scheduling, locks)
- Uses CPU wait cycles
  - Cache miss, dependences, wrong branch prediction
- If one logical CPU uses FP unit, then the other one can
  use INT unit
- Benefits depend on the applications

**Pentium 4 Hyperthreading**

- Duplicated (kahdenennettu)
  - IP, EFLAGS and other control registers
  - Instruction TLB
  - Register renaming logic
- Split (puolittu)
  - No monopoly, non-even split allowed
  - Reordering buffers (ROB)
  - Micro-op queues
  - Load/store buffers
- Shared (puolitettu)
  - Register files (128 GPRs, 128 FPRs)
  - Caches: trace cache, L1, L2, L3
  - Registers needed during µops execution
  - Functional units: 2 ALU, 2 FPU, 2 Id/tst-units

**Core i7** (Nov 2008)

- Intel Nehalem arch.
- 8 cores on one chip, 1-16 threads (820 million transistors)
- Core i7 (Nov 2008)
Superscalar ARM CORTEX-A8

- In family of ARM “application processors”
- Embedded processor running complex operating system
  - Wireless, consumer and imaging applications
  - Mobile phones, set-top boxes, gaming consoles, automotive navigation/entertainment systems
- Three (four?) functional units
  - Fetch pipeline, decode pipeline, execute pipeline
  - SIMD pipeline NEON (10-stages)
- Dual, in-order-issue, 13-stage pipeline
  - Keep power required to a minimum
  - Out-of-order issue would need more logic consuming more power

ARM Cortex-A8 Instruction Fetch Unit

- Predicts instruction stream
- Fetches instructions from the (included) L1 instruction cache
  - Into buffer for decode pipeline
  - Up to four instructions per cycle
- Speculative instruction fetches
- Branch or exceptional instruction cause pipeline flush
- Two-level global history branch predictor
  - Branch Target Buffer (BTB) and Global History Buffer (GHB)
- Return stack to predict subroutine return addresses
- Can fetch and queue up to 12 instructions

ARM Cortex-A8 Instruction Decode Unit

- Dual pipeline structure, pipe0 and pipe1
  - Two instructions at a time
  - Pipe0 contains older instruction in program order
  - If instruction in pipe0 cannot issue, pipe1 will not issue
- In-order instruction issue and retire
  - Results written back to register file at end of execution pipeline
  - no WAR hazards
  - tracks WAW hazards and straightforward recovery from flush
  - Decode pipeline to prevent RAW hazards

ARM Cortex-A8 Instruction Fetch Unit Processing Stages

- F0 address generation unit (AGU)
  - Next address sequentially
  - Or branch target address (from branch prediction of previous address)
- F1 fetch instructions from L1
  - In parallel, check the branch prediction for the next address
- F2 Place instruction to instruction queue
  - If branch prediction, new target address sent to AGU
- Issues instructions to decode two at a time

ARM Cortex-A8 Instruction Decode Unit Processing Stages

- D0 Decompress Thumbs and do preliminary decode
- D1 Instruction decode completed
- D2 Write/read instructions to/from pending/replay queue
- D3 instruction scheduling logic
  - Scoreboard predicts register availability using static scheduling
  - Dependency checking
- D4 Final decode: control signals for integer execute load/store units
**ARM Cortex-A8 Integer Execution Unit**
- Two symmetric (ALU) pipelines, an address generator for load and store instructions, and multiply pipeline.
- Multiply unit instructions routed to pipe0.
  - Performed in stages E1 through E3.
  - Multiply accumulate operation in E4.
- E0 Access register file
  - Up to six registers for two instructions.
- E1 Barrel shifter if needed.
- E2 ALU function
  - E3 If needed, completes saturation arithmetic.
- E4 Change in control flow prioritized and processed.
- E5 Results written back to register file.

**ARM Cortex-A8 Load/Store Pipeline**
- Parallel to integer pipeline.
- E1 Memory address generated from base and index register.
- E2 address applied to cache arrays.
- E3 load, data returned and formatted.
- E3 store, data are formatted and ready to be written to cache.
- E4 Updates L2 cache, if required.
- E5 Results are written to register file.

**ARM Cortex-A8 NEON & Floating Point Pipeline**

**Summary**
- What does superscalar mean?
- ILP vs. machine level parallelism?
- Dispatch, issue, window of execution
- Out-of-order completion
- New dependencies and solutions for them?
- Renaming, solution for name dependencies
- Superscalar Pentium and ARM.
Review Questions

- Differences / similarities of superscalar and traditional pipeline?
- What new problems must be solved?
- How to solve those?
- What is register renaming and why it is used?