The R10000 Superscalar

Mark Greenstreet

CpSc 448B – Oct. 11, 2011

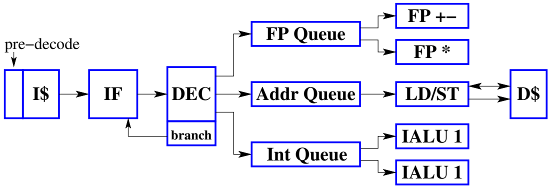
Outline:

- DLS Announcement
- The R10000 Superscalar

DLS Announcement

- The October 13 lecture will be Maurice Herhily's *Distinguished Lecture Series* talk:
  - **Multicore, Transactions, And The Future Of Distributed Computing**
  - Dempster, room 110.
  - Thursday, October 13, 15:30-17:00.
- The content of the Herhily's talk will be included on the midterm.
  - I intend to ask an easy question equivalent to "Did you go to the talk and pay attention."
  - If you go to the talk and pay attention, the answer to the midterm question will be obvious.
  - If for some unavoidable reason you can't go to the talk, the video for the talk will be on the CS department web, and I'll add a link from the course website.
  - **Note:** I find that attending the live lecture is much more effective than watching a video on my computer.

The R10000 Superscalar processor

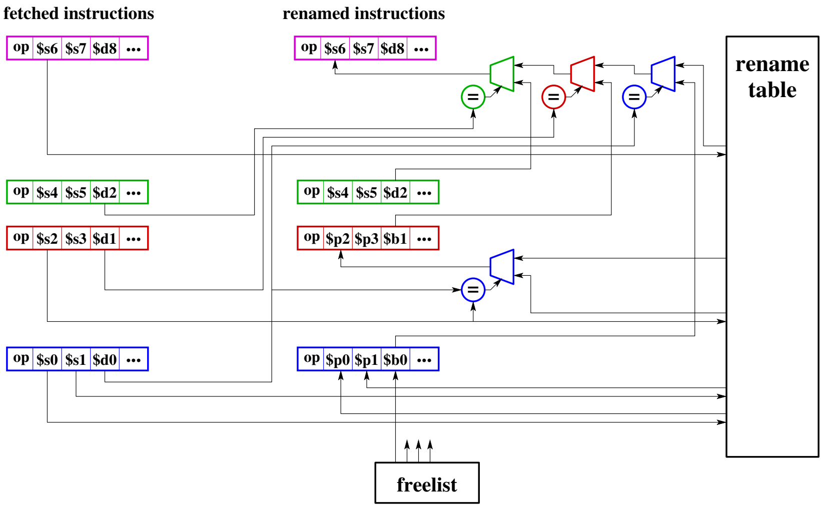


- Superscalar, out-of-order execution.
  - Fetch and rename
  - Ready bits.
  - Branches and exceptions.
- Memory issues
  - stores
  - loads
  - caches

Fetch

- Fetch: four instructions per cycle at arbitrary location in 16 word cache line.
  - the extra logic to allow arbitrary alignment means the compiler doesn't need to worry to make loops aligned on 4-instruction boundaries.
  - Care still needed when crossing cache line boundaries.
    - A good compiler will make sure that a critical loop is aligned to avoid crossing extra cache-line boundaries.
  - See Figure 4 from the paper for details.

Rename (1/2)


fetched instructions
renamed instructions
op $s6 $s7 $d8 ...
op $s6 $s7 $d8 ...
rename table
op $s4 $s5 $d2 ...
op $s4 $s5 $d2 ...
op $s2 $s3 $d1 ...
op $p2 $p3 $b1 ...
op $s0 $s1 $d0 ...
op $p0 $p1 $b0 ...
=
=
=
=
freelist

Rename (2/2)

- On the previous slide, I omitted lots of details to simplify the figure.
  - many connections the the rename table and freelist wire not shown.
  - Likewise, I only showed the comparisons needed for two out of six source registers than need them.
  - I didn't show the logic for updating the rename table and freelist.
- If a processor decodes I instructions in a cycle, and each instruction reads R registers and writes W registers.
  - Then we need a total of

    $$(R+W)W\sum_{J=0} I-1J \quad = \quad (R+W)W\frac{I(I-1)}{/}2$$

    comparators.
  - For the R10000, $I=4$, $R=2$, and $W=1$.
    - ★ I get that it needs 18 comparators.
    - ★ The papers says 24 were used.
    - ★ I'm not sure what the extra 6 are used for.

Ready Bits

- When an instruction is inserted into an issue queue:
  - The ready/busy status of each register that it reads is recorded.
  - These bits are updated as other instructions write their results to the register file.
    - The R10000 can write up to three registers per cycle.
    - Thus, each issue-queue entry requires three comparators.
    - The R10000 has three, 16-entry issue queues.
    - Thus, it needs 48 more comparators to track busy-bits.
    - This is another place where design complexity grows quadratically with issue width.
- When an instruction is ready to execute
  - eligible instructions are selected in a round-robin fashion.
  - some instructions, such as branches are given higher priority.
  - the instruction reads its registers when it issues.
    - The integer register file requires 7 read ports, and three write ports.
    - Register file area grows quadratically with the number of ports.
    - Yet another place that design complexity grows quadratically with issue width.

Branches

- When a branch is encountered in the decode stage
  - The branch outcome is predicted based on the branch history.
  - The instructions fetched in that cycle are discarded if the branch is predicted as taken.
  - The current register mappings are copied into an entry in the branch stack.
  - Subsequent instructions are marked as depending on this branch.
- When a branch is executed.
  - If the prediction was correct
    - All instructions that depended on the branch have their dependence-bit cleared.
    - The branch-stack entry for the branch is reclaimed.
  - If the prediction was incorrect
    - All instructions that depended on the branch are aborted.
    - The register mappings are restored to what they were before the branch.
    - Execution resumes on the correct path.

Exceptions

- Exceptions are raised when the faulting instruction is ready to graduate.
  - This ensures that exceptions occur in program order.
- The CPU maintains an “active list” of instructions that have been issued but not yet graduated.
  - When an exception occurs, this list is “unwound” from the last instruction to issue back to the instruction that raised the exception.
  - As the list is unwound, register mappings are restored.
  - When the excepting instruction is unwound, the CPU has same state as it did just before decoding that instruction.
  - Now, it’s ready to handle the exception.

Stores

Loads

Caches

Hypercube – how big are they?