Reorder Buffer Implementation

Hardware data structures

- **retirement register file (RRF)**
  (~ IBM 360/91 physical registers)
  - physical register file that is the same size as the architectural registers
  - holds values of committed instructions

Reorder Buffer Implementation

Hardware data structures

- **reorder buffer (ROB)**
  (~ R10K active list)
  - provides in-order instruction commit
  - circular queue with head & tail pointers
  - holds 40 “executing” instructions in program order (dispatched but not yet committed)
  - field for either integer or FP result after it has been computed
  - a result value is put in its register in the RRF after its producing instruction has committed (i.e., reaches the head of the buffer & is removed)

Reorder Buffer Implementation

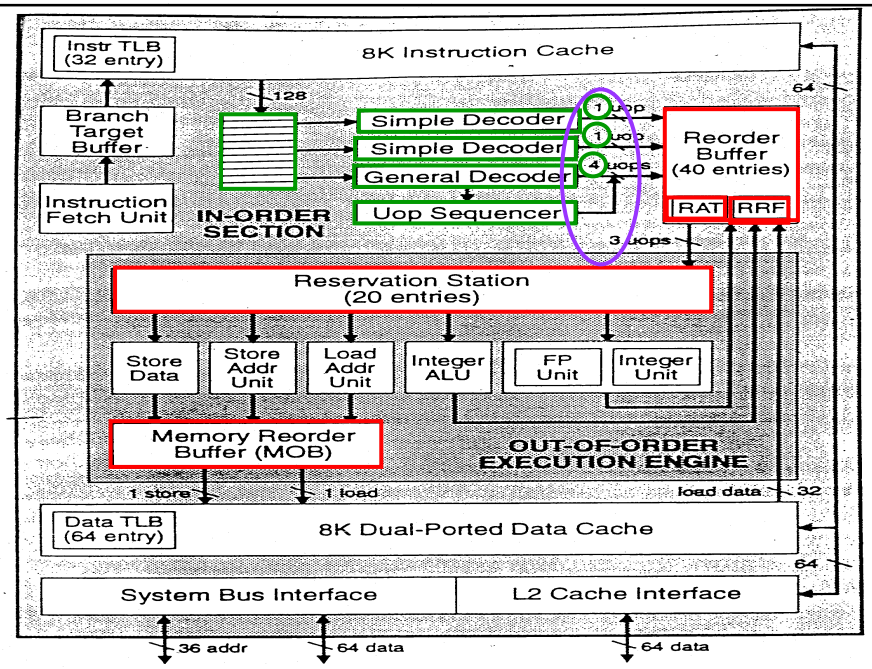
Hardware data structures

- **register alias table (RAT)**
  (~ R10K map table)
  - provides register renaming
    - important because very few GPRs in the x86 architecture
  - indicates whether a source operand of a new instruction points to the reorder buffer or the physical register file
  - do an associative search of ROB destination registers for the new source operands
    - if found, consumer instruction points to the producer instruction in the ROB
    - the data hazard check before instruction dispatch

Reorder Buffer Implementation

Hardware data structures

- **reservation station**
  (~ IBM 360/91 reservation stations, R10000 instruction queues)
  - holds instructions waiting to execute
  - provides forwarding to reduce RAW hazards
    - result values go back to the reservation station (as well as ROB) so dependent instructions have source operand values
  - provides out-of-order execution



Pentium Execution

In-order issue

- decode instructions
- rename registers via register alias table
- enter **uops** into reorder buffer for in-order completion
- detect structural hazards for reservation station

Out-of-order execution

- one reservation station, multiple entries
- check source operands for RAW hazards
- check structural hazards for separate integer, FP, memory units
- execute instruction
- result goes to reservation station & reorder buffer

In-order commit

- this & previous uops have completed
- write “G”PR registers
- rollback on interrupts

Pentium

fetch & decode pipeline

- BTB access (1 stage)
- instruction fetch & align for decoding (2.5 stages)
- decode & uop generation (2.5 stages)
- **register renaming & instruction issue to reservation stations** (3 stages minimum)

integer pipeline

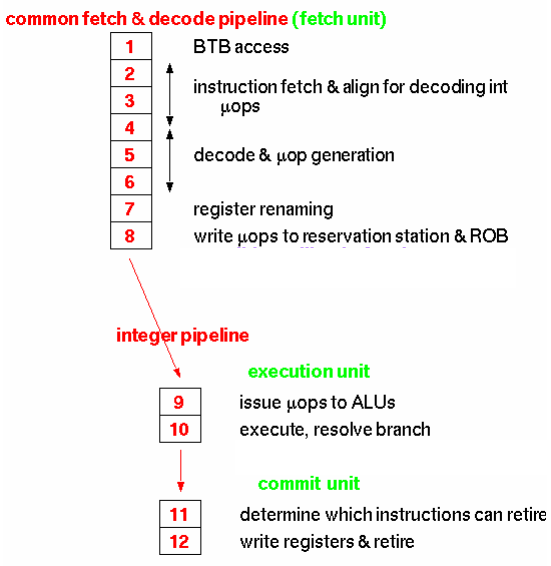
- execute, resolve branch
- write registers & commit

load pipeline

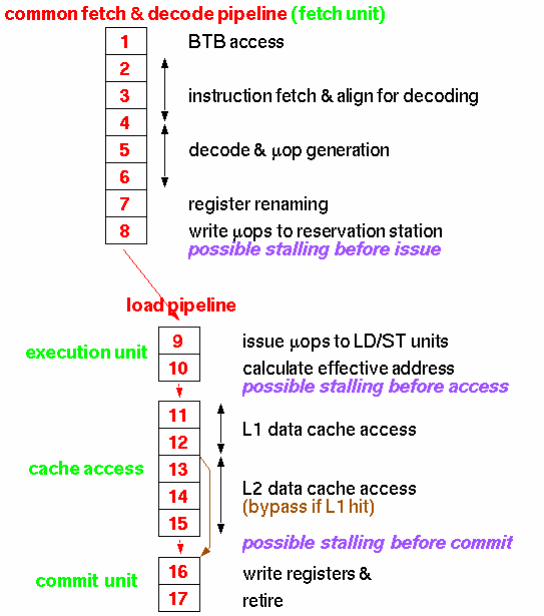
- address calculation & to memory reorder buffer
- integrated L1 & L2 data cache access

pipelined FP add & multiply

Pentium P6



Pentium P6



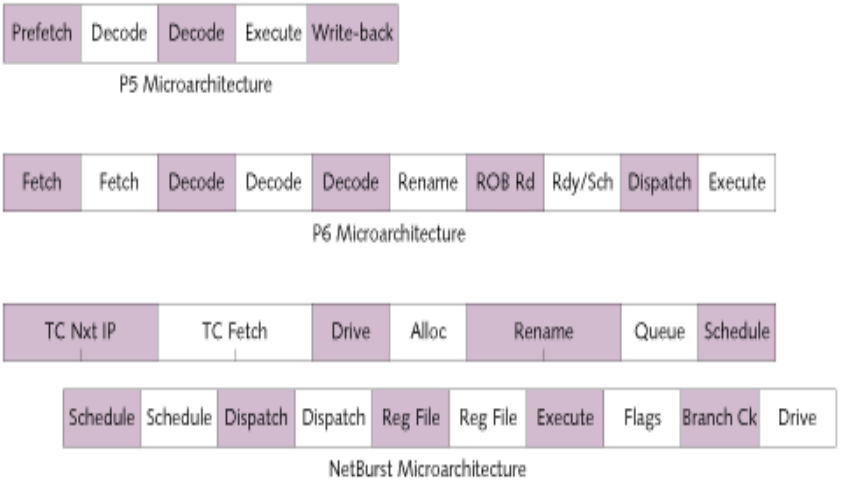
Pentium 4 (Netburst)

What the Pentium 4 looks like to me

Adopts the physical register file model

- **physical register file**
  - 128 entries for both committed & in-flight operands
- **ROB**
  - operand status only
  - also 128 entries
- **2 register alias tables**
  - retirement RAT for operands of committed instructions
  - front-end RAT for operands of in-flight instructions

Pentium Pipeline Comparisons



Pentium 4

Some bandwidth constraints: maximum for one cycle

- 16 bytes fetched
- 3 instructions decoded
- 6 μops issued to the reorder buffer
- 4 μops dispatched to reservation station & functional units
- 1 load & 1 store access to the L1 data cache
- 1 cache result returned
- 3 μops committed

if

- good instruction mix
- good instruction order
- operands available
- functional units available
- load & store to different cache banks
- all previous instructions already committed

Pool of Physical Registers vs. Reorder Buffer

Think about the advantages and disadvantages of these implementations

- book claims that physical register commit is simpler
  - record that value no longer speculative in register busy table
  - unmap previous mapping for the architectural register
- instruction issue simpler (physical register pool)
  - only look in one place for the source operands (the physical register file)
- book claims that deallocating register is more complicated with a physical register pool
  - have to search for outstanding uses in the active list
  - but not done in practice: wait until the instruction that redefines the architectural register commits
- faster to index map table to get source operands than do associative search on ROB
  - can have more outstanding results

Limits

Limits on out-of-order execution

- amount of ILP in the code
- scheduling window size
  - need to do associative searches & its effect on cycle time
  - relatively few instructions in window
- number & types of functional units
- number of locations for values
- number of ports to memory