# Super-Scalar CPUs

- # execution units (INT/FP)
  - Pentium (2/1), P6 (2+/2), P7 (4/2)
  - 21164 (2/2), 21264 (4/2)
- Units often specialised e.g 21264:
  - Int ALU + multiply
  - Int ALU + shifter
  - 2x Int ALU + load/store
  - FP add + divide + sqrt
  - FP multiply

- Max issue rate
  - Pentium (2), P6&P7 (3$\mu$ops)
  - 21164 (4), 21264 (4)
  - Ideal instruction sequence
    - Right combination of INT and FP
  - Lower than number of exec units
- Two basic types
  - Static in-order issue (21164, Pentium, Merced)
  - Dynamic out-of-order execution (21264, PPro)

# Static Scheduled

- All instructions begin execution in-order

1. Fetch and decode a block of instructions

2. 'Slot' instructions to pipes based on function unit capability and current availability

3. Issue checks:

   - Data Hazards
     - Are the instructions independent?
     - Check register scoreboard
       * Are the source operands ready?

      * Will write order be preserved?

   – Non-blocking missed loads

      * Do not stall until value is used

   – *Maintain in-order dispatch*

- Control hazards

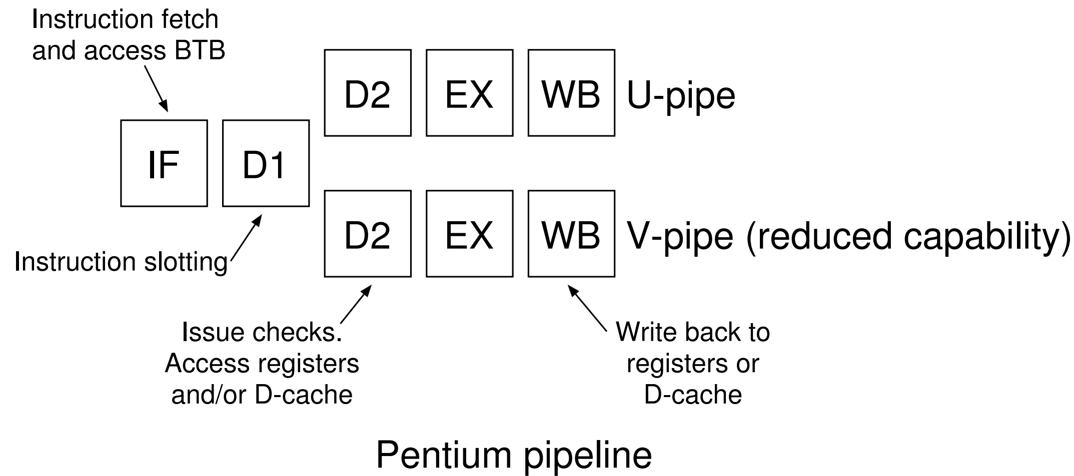
   – Is one of the instructions a predicted-taken branch?

      * Discard instructions past branch

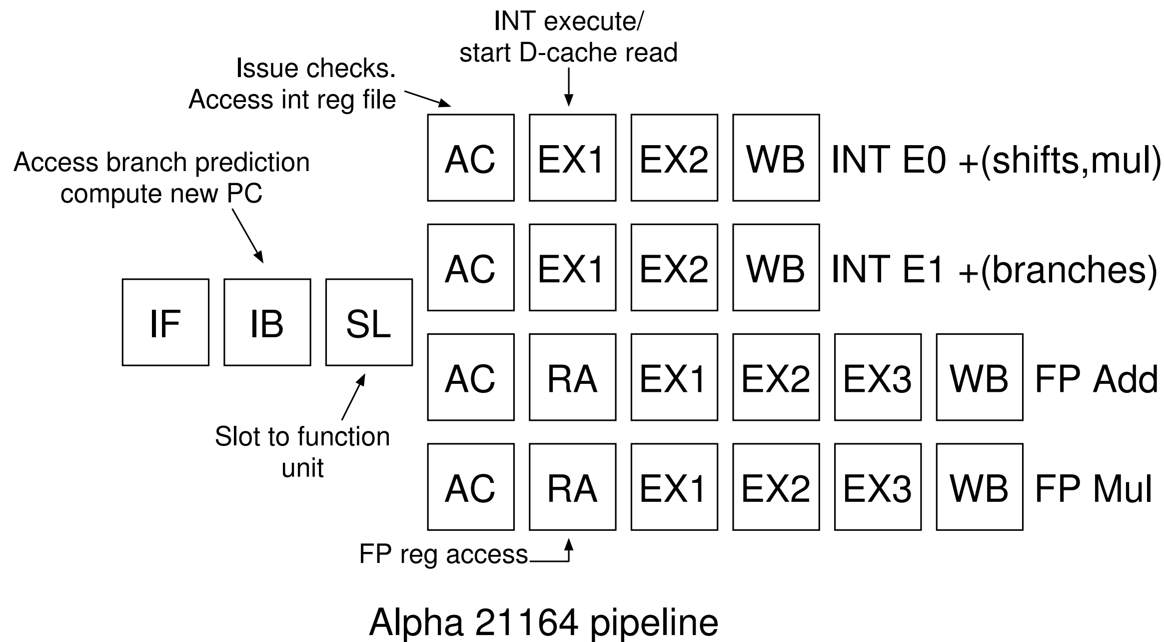
   – Be prepared to squash speculated instrs

4. move onto next block when all issued

# Static Scheduled Examples




Pentium pipeline




Alpha 21164 pipeline

# Static Scheduled Super-Scalar

- Relies greatly on compiler
  - Instruction scheduling
    - slotting
    - data-dependence
  - Issue loads early (or prefetch)
  - Reduce # branches and jumps
    - unroll loops
    - function inlining
    - use of CMOV/predication
  - Align branches and targets

  * avoid wasted issue slots

- Optimization can be quite implementation dependent

- Static analysis is imperfect
  - basic blocks can be reached from multiple sources
  - compiler doesn't know which loads will miss
  - Feedback Directed Optimization can help

$\Rightarrow$ On most code, actual issue rate will be $<<$ max

# Helping the compiler

- Wish to issue loads as early as possible, but
  - mustn't overtake a store/load to same address
  - Stack / Global variables solvable
    - `[r12,4] != [r12,16]`
  - Heap refs harder to disambiguate
    - `[r2,8] != [r4,32] ???`
    - `C/C++` particularly bad in this respect

⇒ Data speculation (IA-64, Transmeta)

  - allows loads to be moved ahead of possibly conflicting load/stores
  - `ld.a r3 = [r5]` enters address into Address Aliasing Table
  - any other memory reference to same address removes entry
  - `ld.c r3 = [r5]` checks entry is still present else reissues load

- Predication enables load issue to be hoisted ahead of branch, but not above compare

⇒ Control speculation (IA-64)

- `ld.s r3 = [r5]` execute load before it is known if it should actually be executed
- `chk.s r3, fixup` check poison bit and branch if load generated an exception

# Dynamic Scheduling

- Don't stop at the first stalled instruction, continue past it to find other non-dependent work for execution units

- Search window into I-stream
  - Data-flow analysis to schedule execution
  - Out-of-order execution
  - In-order retirement to architectural state
  - P6 core $\leq$ 30 $\mu$ops, P7 $\leq$ 126

- Use *speculation* to allow search window to extend across multiple basic blocks
  - (Loops automatically unrolled)

- Need excellent branch prediction
- Track instructions so they can be aborted if prediction was wrong
- Try to make branch result available ASAP (to limit waste caused by mis-prediction)

# Register Renaming

- Register reuse causes false dependencies

- (Often referred to as name-dependencies)
  - WaR, WaW: no data transfer

- Undo compiler's register colouring

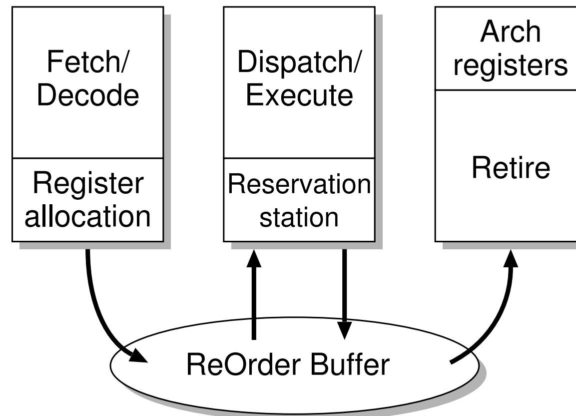
- Necessary to unroll loops

⇒ Register renaming
  - Large pool of internal physical registers
  - P6 40, 21264 80+72, P7 128

- New internal register allocated for the result operand of every instruction
- Re-mapper keeps track of which internal registers instructions should use for their source operands
  * needs to be able to rollback upon exception or mispredict

- Architectural register state updated when instructions retire

# Out-of-Order Execution




1. Fetch and decode instructions

2. Re-map source operands to appropriate internal registers. Allocate a destination register from register free list. Place instruction in a free Re-Order Buffer (ROB) slot.

3. Reservation station scans ROB to find instructions for which all source operands are available, and a suitable execution unit is free

(Favour older instructions if multiple ready)

4. Executed instructions and results are returned to the ROB (internal registers which are no longer needed are placed on free list)

5. Retire unit removes completed instructions from ROB in-order, and updates architecturally visible state. Detect exceptions & mis-predicted branches; Roll-back ROB contents and mapping register state, start fetch from new PC

# Loads and Stores

- Dyn Exec helps hide latency of L2/L3 cache
  - Find other work to do in the meantime
  - Allow loads to issue early

- Stores cannot be undone

  $\Rightarrow$ Update memory in Retirement stage
  - Hold in Store Queue until retirement

- Loads that overtake stores must be checked to see if they refer to the same location (alias)
  - Address of store may not yet be known

⇒ Speculate load and check later:

- Load Queue stores addresses of issued loads until they retire
- when a store ‘executes’ (target address is known) it checks the LQ to see whether a newer load has issued to the same address
- if so, execution is rolled back to the store instruction (replay load)
  * 21264 has 32 entry LQ and a 1024 entry prediction cache to predict which loads to ‘hold back’ and thus avoid replay trap

- Loads overtaking loads treated similarly to maintain ordering rules with other CPUs/DMA

# Out-of-order Execution (2)

- Tomasulo's algorithm used on IBM360/91.
- Less dependency on compiler than static-sched
- Better at avoiding wasted issue slots
- But, O-o-O execution uses a lot of transistors
  - ReOrder Buffer and Reservation Stations are large structures incorporating lots of Content Addressable Memory
  - Tend to be at least $O(N^2)$ in complexity
  - Tend to be on critical path
    - diminishing returns...
  - 20%+ of chip area on 21264
- Factors effecting usable ILP
  - Window size
  - Number of renamed registers
  - Memory reference alias analysis
  - Branch and jump prediction accuracy

  - Data cache performance
  - (Value speculation performance)

- Simulation suggests the ‘perfect’ processor: 18-150 instructions per cycle for SPEC92

- 10 way for int progs feasible, more for FP

- Some code just exhibits very poor ILP...

# VLIW Architectures

- Very Long Instruction Word (VLIW)

- Each instruction word (or ’packet’) contains fields to specify an operation for each function unit

- Compiler instruction scheduling:

  - allocates sub-instructions to function units

  - avoids any resource restrictions

  - ensures producer-consumer latencies satisfied (delay slots)

✔ CPU doesn’t need to worry about issue-checks

⇒ High clock speed

✘ Relies heavily on compiler / assembler programmer

- loop unrolling
- trace scheduling

✘ Stall in any function unit causes whole processor to stall

- D-cache misses a big problem

✘ Often sparse I-stream (lots of nops)

✘ Exposes processor internals

- Typically no binary compatibility

# Intel EPIC (VLIW-like)

- Intel: Explicitly Parallel Instruction Computer
  - Merced (Itanium) , McKinley

- Three 41 bit instrs packed into 128 bit 'bundle' with 5 template bits

- Template determines function unit dispatch
  - restricted set of possibilities simplifies instruction dispersement hardware
    - e.g. `[Mem,Int,Branch]`, no `[Int,Int,Int]`

- Stop bits: barriers between independent instructions groups

- groups can cross multiple bundles

- Compiler collects instrs into independent groups
- Hardware interlock of longer-latency instructions as well as load-use latencies

✔ Reduces issue-check complexity for CPU

✔ Retains binary compatibility

- Need good compilers
  - hope extensive use of load speculation instructions enables hoisting of loads to avoid stalling whole CPU
- Optimization for new implementations important?

# Advantages of HW (Tomasulo) vs. SW (VLIW) Speculation

HW advantages:

- Better at memory disambiguation since knows actual addresses
- HW better at branch prediction since lower overhead
- HW maintains precise exception model
- HW does not execute bookkeeping instructions
- Same software works across multiple implementations
- Smaller code size (not as many nops filling blank instructions)

SW advantages:

- Window of instructions that is examined for parallelism much higher
- Much less hardware involved in VLIW (unless you are !)
- More involved types of speculation can be done more easily
- Speculation can be based on large-scale program behavior, not just local information

Taken from Patterson cs252.

# Transmeta ‘Code Morphing’

- VLIW core hidden behind x86 emulation
  - VLIW with in-order execution
  - 64 Integer registers
  - 32 floating point registers
  - Simple in-order, 6-stage integer pipeline:
  - 2 fetch stages, 1 decode, 1 register read,
  - 1 execution, and 1 register write-back
  - 10-stage pipeline for floating point, which has 4 extra execute stages
  - Instructions in 2 sizes: 64 bits (2 ops) and 128 bits (4 ops)
- Uses combination of interpretation, translation and on-line feedback-directed optimization
- Only ‘code morphing’ s/w written for VLIW
  - Apps, OS and even BIOS are x86
- Keeps an in-memory translation cache

- Translate and optimise along frequently executed paths (trace scheduling)
  - speculative load instrs increase trace length
- Hardware features to assist translation:
  - Shadow registers with commit instruction
    - assist rollback upon x86 exceptions/mispredicts
  - hold-back stores until commit
- Performance counters assist re-optimization

✔ Binary compatibility, High clock speed, Low power

✔ Potential for more complex scheduling than h/w

✘ Overhead of performing translation

✘ Less dynamic than h/w scheduling