

---

# Instruction Fetch Microarchitecture

Wen-mei Hwu

ECE

University of Illinois,  
Urbana-Champaign

---

# Instruction Fetch

- Underlying principle
  - Fetch rate must match or exceed expected execution rate
- Strategy: in every clock cycle
  - pull in enough bytes from I-cache
  - Decide what the next instructions are with branch prediction

---

# Why Branch Prediction

- simple pipelines (fig.)
  - branches executed at later stage
  - most RISCs stage 2 or 3
  - more for CISCs - longer decode
- out-of-order pipelines (fig.)
  - branch often executed long after fetch (data dependence delay).

---

# Branch Prediction Issues

- Conditional branches
  - address prediction
  - direction prediction
- Unconditional branches
  - indirect branches
  - return instructions

---

# 2-Level Adaptive Br. Pred.

- Branch history pattern to index
  - a table of traditional 2-bit counters
- Decision on 2-level format (fig.)
  - global/set/private history pattern registers
  - global/set/private pattern tables

---

# Speculative History Update

- Tight loops require up-to-date branch history (fig.)
  - true branch direction not known yet.
  - Several instances of a branch in the pipeline.

---

# Out-of-order History Update

- Branches can execute out of order
  - allow fast execution branches to speculatively confirm pattern
  - must repair history when mispredict

---

# Branch Clustering

- Multiple branches contending for one BTB set
  - due to ignored lower order address bits when indexing BTB
  - more common in variable I-formats
  - set associativity helps

---

# Return Stacks

- Return instructions are different
  - return address is different for each caller function
- Solution - return stack

---

# Wide Fetch Design

- Goal
  - pulling in enough instructions to match machine issue width
- Issues
  - I-cache alignment
  - branch prediction adaptation

---

# Complications

- Wide issue processors require fetching across
  - multiple cache blocks
  - short distance predict taken branch
  - long distance predict taken branch
  - back edge branches

---

# Current Solutions

- Large cache blocks relative to fetch size
  - UltraSPARC and P6
  - Hope that most fetches are contained in one cache block

---

# Current Solutions (cont.)

- Cache banking (fig.)
  - Pentium and K5
  - allows sequential fetch across cache blocks during same cycle
  - realignment hardware
  - still need large blocks for higher issue rate

---

# Cache Banking

- Enable access to the next block
  - detection of fetch starting at the second part of a block
  - additional tag check to ensure hit in next block

---

# Cache Banking (cont.)

- Alignment of bytes to decoder
  - must select bytes from bank 0 and 1 into decoder byte position
- Avoid bank conflict
  - cache block size should be at least twice the fetch size

---

# Advanced Wide Fetch

- Short dist. predict taken branch
  - mask
- Long dist. Predict taken branch
  - non-sequential set access and mask
- Loop back branches
  - replication of I-cache output bytes

---

# Predicting Multiple Branches

- Predict target cache block (1st)
  - combined effect of all currently fetched branches
  - limited by target addresses available from each BTB access
    - avoid storing targets of unlikely branches in BTB

---

# Predicting Multiple Branches (cont.)

- Select bytes in target cache block (2nd)
- Must remove bottlenecks elsewhere
  - branch decoders
  - branch execution units
  - retirement bandwidth

---

# Instruction Decoding

- Source of complexity
  - Variable instruction format
    - variable number of operands
    - variable sized operand specifier
  - Prefixed instructions
    - used for opcode space extension

---

# Pre-Decoding

- Instruction boundaries
  - needed for parallel decoding
  - mark start/end bytes of instructions
- Instruction steering
  - directing instructions to decoders
  - digest prefixes

---

# Actual Decoding

- Decoder strength
  - some decoders may not handle all instruction types
  - instructions may be delayed if required decoder not available
- Very complex instructions
  - microsequencer and microcode

---

# Performance Issues

- Strength of decoders
  - X86 has 16-bit and 32-bit instruction formats
  - 32-bit format tend to require less decoding logic
  - How important is 16-bit code?

---

# Performance Issues

- Branch decoding
  - target address calculation units
  - verify branch prediction for unconditional branches and jump
    - allow decoder to signal prediction miss early
    - reduced prediction miss penalty

---

# Alternative Strategies

- Predecoded Instruction Cache
  - mark bytes while loading I-cache
    - start/end byte positions of instructions
    - opcode and prefix locations
- Advantage
  - reduced number of decode stages

---

# Alternatives (cont.)

- Disadvantages
  - cache loading process must understand the state of length decoder to handle instructions that span cache blocks
  - longer refill on a cache miss
  - enlarged cache structure

---

# Alternatives (cont.)

- Pipelined decode
  - predecode after fetch from cache
- Advantage
  - simplifies I-cache design
- Disadvantages
  - longer pipeline, must have better prediction

---

# Decoding Examples

- Intel P6
  - up to 3 X86 instructions
  - 3 decoders, deliver 4,1,1 micro-ops
  - 6 micro-ops max total
  - 4 micro-ops per cycle for microcode
  - issues with 16-bit code

---

# Decoding Examples (cont.)

- AMD k6
  - up to 2 X86 instructions
  - 2 full decoders
  - 4 micro-ops max total
- Cyrix M1
  - up to 2 X86 instructions
  - 2 Full decoders - no micro-ops

---

# Instruction Buffering

- De-couple fetch from decode
  - sometimes decoding takes long
    - microsequenced instructions
  - sometimes decoding stalls
    - insufficient decoder strength

---

# Self-modifying Code

- Detect with I-cache snooping
  - must ensure that all instructions in pipeline are also in I-cache
  - cache line replacement
    - large victim cache to ensure coverage
- More in X86 than others