CSE 502: Computer Architecture

Superscalar Decode
Superscalar Decode for RISC ISAs

- Decode $X$ insns. per cycle (e.g., 4-wide)
  - Just duplicate the hardware
  - Instructions aligned at 32-bit boundaries
uop Limits

• How many uops can a decoder generate?
  – For complex x86 insts, many are needed (10’s, 100’s?)
  – Makes decoder horribly complex
  – Typically there’s a limit to keep complexity under control
    • One x86 instruction → 1-4 uops
    • Most instructions translate to 1.5-2.0 uops

• What if a complex insn. needs more than 4 uops?
UROM/MS for Complex x86 Insts

- UROM (microcode-ROM) stores the uop equivalents
  - Used for nasty x86 instructions
    - Complex like > 4 uops (PUSHA or STRREP.MOV)
    - Obsolete (like AAA)

- Microsequencer (MS) handles the UROM interaction
UROM/MS Example (3 uop-wide)

Cycle 1
- ADD
- SUB
- STORE
- REP.MOV
- ADD [ ]
- STD
- SUB
- STA
- LOAD
- XOR
- ADD
- SUB
- LOAD

Cycle 2
- REP.MOV
- ADD [ ]
- XOR
- REP.MOV
- ADD [ ]
- XOR
- ADD [ ]
- XOR
- XOR

Cycle 3
- INC
- STORE
- mJCC
- LOAD
- XOR
- INC
- mJCC
- XOR
- XOR

Cycle 4
- LOAD
- mJCC
- ADD

Cycle 5
- XOR

Cycle n
- ... (repeated)

Cycle n+1
- XOR
- mJCC
- LOAD
- ADD

Complex instructions, get uops from mcode sequencer

Fetch - x86 insts
Decode - uops
UROM - uops
Superscalar CISC Decode

- Instruction Length Decode (ILD)
  - Where are the instructions?
    - Limited decode – just enough to parse prefixes, modes
- Shift/Alignment
  - Get the right bytes to the decoders
- Decode
  - Crack into uops

Do this for N instructions per cycle!
ILD Recurrence/Loop

\[ \text{PCI} = X \]
\[ \text{PCI} + 1 = \text{PCI} + \text{sizeof( Mem[PCI] )} \]
\[ \text{PCI} + 2 = \text{PCI} + 1 + \text{sizeof( Mem[PCI+1] )} \]
\[ = \text{PCI} + \text{sizeof( Mem[PCI] )} + \text{sizeof( Mem[PCI+1] )} \]

• Can’t find start of next insn. before decoding the first
• Must do ILD serially
  – ILD of 4 insns/cycle implies cycle time will be 4x
Bad x86 Decode Implementation

ILD dominates cycle time; not scalable
Hardware-Intensive Decode

- Decode from every possible instruction starting point!
- Giant MUXes to select instruction bytes
ILD in Hardware-Intensive Approach

Previous: $3 \times \text{ILD} + 2 \times \text{add}$

Now: $1 \times \text{ILD} + 2 \times (\text{mux} + \text{add})$

Total bytes decode = 11
Predecoding

- ILD loop is hardware intensive
  - Impacts latency
  - Consumes substantial power
- If instructions A, B and C are decoded
  - ... lengths for A, B, and C will still be the same next time
  - No need to repeat ILD

Possible to cache the ILD work
Decoder Example: AMD K5 (1/2)

From Memory
8 bytes
Predecode Logic
13 bytes
I$

8 bits
b0 b1 b2 ... b7
+5 bits
b0 b1 b2 ... b7

8 × (8-bit inst + 5-bit predecode)

Decode

Up to 4 ROPs

8 × (8-bit inst + 5-bit predecode)

Compute ILD on fetch, store ILD in the I$
Decoder Example: AMD K5 (2/2)

• Predecode makes decode easier by providing:
  – Instruction start/end location (ILD)
  – Number of ROPs needed per inst
  – Opcode and prefix locations

• Power/performance tradeoffs
  – Larger I$ (increase array by 62.5%)
    • Longer I$ latency, more I$ power consumption
  – Remove logic from decode
    • Shorter pipeline, simpler logic
    • Cache and reused decode work → less decode power
  – Longer effective I-L1 miss latency (ILD on fill)
Decoder Example: Intel P-Pro

- Only Decoder 0 interfaces with the uROM and MS
- If insn. in Decoder 1 or Decoder 2 requires > 1 uop
  1) do not generate output
  2) shift to Decoder 0 on the next cycle
Fetch Rate is an ILP Upper Bound

• Instruction fetch limits performance
  – To sustain IPC of N, must sustain a fetch rate of N per cycle
    • If you consume 1500 calories per day, but burn 2000 calories per day, then you will eventually starve.
  – Need to fetch N on average, not on every cycle
• N-wide superscalar ideally fetches N insns. per cycle
• This doesn’t happen in practice due to:
  – Instruction cache organization
  – Branches
  – ... and interaction between the two
Instruction Cache Organization

- To fetch $N$ instructions per cycle...
  - L1-I line must be wide enough for $N$ instructions
- PC register selects L1-I line
- A *fetch group* is the set of insns. starting at PC
  - For $N$-wide machine, $[\text{PC}, \text{PC}+N-1]$
Fetch Misalignment (1/2)

- If PC = xxx01001, N=4:
  - Ideal fetch group is xxx01001 through xxx01100 (inclusive)
Now takes two cycles to fetch $N$ instructions

- $\frac{1}{2}$ fetch bandwidth!

Might not be $\frac{1}{2}$ by combining with the next fetch
Reducing Fetch Fragmentation (1/2)

• Make $|\text{Fetch Group}| < |\text{L1-I Line}|$

Can deliver $N$ insns. when $PC > N$ from end of line
Reducing Fetch Fragmentation (2/2)

• Needs a “rotator” to decode instructions in correct order