Modern processors

• Modern superscalar architecture “executes N instructions every cycle”
• Pentium: N = 2 (U-pipe and V-pipe)
• Pentium II+: N=5; dynamic translation to 1-4 µops
• Reality check: ~1.2 instructions/cycle — processor resources are mostly wasted even with good ordering

• Processor spends a lot of time waiting:
  – Memory stalls
  – Branch stalls
  – Expensive arithmetic operations
  – Resource conflicts

Instruction scheduling

• Instruction order has significant performance impact on some modern architectures
• Avoiding stalls requires understanding processor architecture(s) (Intel Arch. SDM Vol. 3, Chapter 13)
• Instruction scheduling: reordering low-level instructions to improve processor utilization

Simplified architecture model

• Assume MIPS-like architecture for illustration — 5 pipeline stages (Pentium II: 9+9)
• F: Instruction fetch -- read instruction from memory, decode
• R: Read values from registers
• A: ALU
• M: Memory load or store
• W: Write back result to registers

Examples

• mov eax, ebx
  F: extract imm. 10
  R: read eax
  A: add operands
  M: read from cache
  W: store into ecx
• add eax, 10
  F: extract imm. 10
  R: read eax
  A: add operands
  M: read from cache
  W: store into ecx
• push [edx + 16]?
Non-pipelined execution

- time
- add eax, 10
- mov ecx, [edx + 16]
- 3-5 cycles per instruction

Pipelined Execution

- time
- New instruction begun every cycle
- Most pipeline stages busy every cycle

Superscalar execution

- Two copies of most execution units
- Many instructions executed in parallel

Memory is slow!

- Main memory: ~100 cycles
- Second-level cache: ~10 cycles
- First-level cache: ~1 cycles
- Loads should be started as early as possible to avoid waiting

Memory Stalls

- mov eax, [ecx + 16]
- add ebx, eax
- memory value available here (if in cache)
- needed here!
- Need to stall processor even if in cache

Solutions:

- Option 1: (original Alpha, 486, Pentium)
  Processor stalls on use of result until available. Compiler should **reorder** instructions if possible:
  - mov eax, [ecx + 16]  mov eax, [ecx + 16]  add ebx, eax  add ecx, 1  add ecx, 1  add ebx, eax
**No interlocks**

- Option 2: (R3000) Memory result not available until two instructions later; compiler must insert some instruction.

  ```
  mov eax, [ecx + 16]
mov ebx, eax
add ecx, 1
  ```

**Out-of-order execution**

- Out-of-order execution (PowerPC, recent Alpha, MIPS, P6): can execute instructions further ahead rather than stall
- Processor can issue instructions farther ahead in stream
- reorder buffer from which viable instructions are selected on each cycle

  ```
  mov ax, [cx + 16]
mov bx, ax
add cx, 1
  ```

**Branch stalls**

- Branch, indirect jump instructions: next instruction to execute not known until address known
- Processor stall of 3-10 cycles!

  ```
  cmp ax, bx
  jz L
  beq r1, r2, L
  ```

**Option 1: stall**

- 80486 stalls branches till pipeline empty!
- Early Alpha processors: start initial pipeline stages on predicted branch target, stall until target address known (3+ cycle stall on branch mispredict)

  ```
  beq r1, L
  mov r3, r4
  ```

**Dealing with stalls**

- Alpha: predicts backward branches taken (loops), forward branches not taken (else clauses), also has branch prediction cache
- Compiler should avoid branches, indirect jumps
  - unroll loops!
  - use conditional move instructions (Alpha, Pentium Pro) or predicated instructions (Itanium) - can be inserted as low-level optimization on assembly code

  ```
  cmp ecx, 16
  jz skip
  mov eax, ebx
  skip:
  ```

**MIPS: branch delay slot**

- Instruction after branch is always executed: branch delay slot

  ```
  beq r1, r2, L
  mov r3, r4
  ```

- Options for compiler:
  - always put nop after branch
  - move earlier instruction after branch
  - move in destination instruction if harmless
- Problem: branch delay slot hasn’t scaled
Other stalls
- Some instructions take much longer to complete—multiply, divide
- Typical superscalar processors: 4-way < 4 copies of some functional units
  - R10000: 2 integer ALU units, 2 floating point ALU units. Pentium: 1/1.
- Issuing too many ALU operations at once ⇒ some pipelines stall
- Interleave other kinds of operations so more pipelines are busy

Real architectures
- Deeper pipelines, superscalar
  - MIPS R4000: 8 stages; R10000: 8 stages × 4 way
  - Alpha: 11 stages, 2 or 4 way
  - Pentium P6 (Pro, II, III): 9 stage (uops), 5 way, 9 stage dynamic translation pipeline
  - Pentium 4: 20-stage branch prediction pipeline, 126 outstanding instructions, 96 outstanding memory ops, deep speculative execution, 256K on-chip 2nd-level cache

Instruction scheduling
- Goal: reorder instructions so that all pipelines are as full as possible
- Instructions reordered against some particular machine architecture and scheduling rules embedded in hardware
- May need to compromise so that code works well on a variety of architectures (e.g. Pentium vs. Pentium II)

Scheduling constraints
- Instruction scheduling is a low-level optimization: performed on assembly code
- Reordered code must have same effect as original
- Constraints to be considered:
  - data dependences
  - control dependences: only reorder within BB
  - resource constraints

Data dependences
- If two instructions access the same register or memory location, they may be dependent
- True dependence: write/read
  \[ \text{mov eax, [ecx + 16]; add ebx, eax} \]
- Anti-dependence: read/write
  \[ \text{add ebx, eax; mov eax, [ecx + 16]} \]
- Output dependence: write/write
  \[ \text{mul ebx; mov edx, ecx} \quad \text{both update edx} \]

Dependence Graph
1. mov ecx, [ebp+8]
2. add ecx, eax
3. mov [ebp + 4], eax
4. mov edx, [ecx + 4]
5. add eax, edx
6. mov [ebp + 4], ebx
Dependence Graph
- Memory dependences are tricky: two memory addresses may be aliases for each other—need alias analysis
  - `mov [edx + 16], eax`
  - `mov ebx, [ecx - 8]`
  - `mov [ecx+8], edx`
- If one instruction depends on another, order cannot be reversed—constrains scheduling
- Any valid ordering must make all dependence edges go forward in code: topological sort of dependence DAG

List Scheduling
- Initialize ready list R with all instructions not dependent on any unexecuted instruction
- Loop until R is empty
  - pick best node in R and append it to reordered instructions
  - update ready list with ready successors to best node
- Works for simple & superscalar processors
- Problem 1: Determining best node in R is NP-complete! Must use heuristic.
- Problem 2: Doesn’t consider all possible executions

Greedy Heuristic
- If instruction predecessors won’t be sufficiently complete yet, creates stall
- Choose instruction that will be scheduled as soon as possible, based on start time of its predecessors: simulate processor
- How to break ties:
  - pick node with longest path to DAG leaf
  - pick node that can go to non-busy pipeline
  - pick node with many dependent successors

Scheduling w/ FRAMW model
- Ready list: 1, 2, 3, 4, 5, 6
- Simulated processor diagram:
  - `mov cx, [bp+8]`
  - `add cx, ax`
  - `mov [bp + 4], ax`
  - `mov [dx + 4], ax`
  - `mov [bp + 4], bx`
- Result: eliminated stalls after 1 & 4, moved memory operations earlier

Register allocation conflict
- Problem: use of same register creates anti-dependencies that restrict scheduling
- Register allocation before scheduling: prevents good scheduling
- Scheduling before register allocation: spills destroy scheduling
- Solution: schedule abstract assembly, allocate registers, schedule again!

Summary
- Instruction scheduling very important for non-fancy processors
- Improves performance even on processors with out-of-order execution (dynamic reordering must be more conservative)
- List scheduling provides a simple heuristic for instruction scheduling