Instruction-Level Parallelism

Kai Shen

Instructional-Level Parallelism

- Instructional-level parallelism
  - A single CPU core, but multiple functional units (arithmetic computation, floating-point computation, …)
  - Possible to execute multiple instructions in parallel
- ⇨ Pipelining: initiate one instruction per cycle
- ⇨ SuperScalar: possibly initiate multiple instructions in a cycle
- Historical perspectives
  - Scalar vs. vector processors ⇨ Superscalar processor
  - RISC/CISC architectures

Sequential Execution Model

- Sequential execution model:
  - One instruction at a time ⇨ all computation completed, all state committed before next instruction starts
- Equivalent executions to sequential execution model:
  - Follow control dependences – execute same set of instructions
  - Follow data dependences – each instruction executed in the same way (using the same input data and producing the same output)

  ⇨ Software written (or compiled) for sequential execution model processors will remain correct on a superscalar processor

Sequential Execution Model: Interrupts

- An interrupt in sequential execution:
  - Instruction at program counter (PC) is interrupted
  - Every instruction before PC has been completed
  - Every instruction after PC has not started
  - Enough is known about PC's execution state so it can be resumed after interrupt
- Precise interrupt:
  - An interrupted state is equivalent to that under sequential execution
  - Why does it matter? – allowing easy state saving and restoration

Manage Dependencies

- Control dependences
  - Not to go beyond a branch instruction until the instruction's outcome becomes available

  ⇨ Execute predicted branch(es) on temporary space and not to commit in case of mis-prediction
- Data dependences
  - Read-after-write, or RAW (true)
  - Write-after-read, or WAR (artificial)
  - Write-after-write, or WAW (artificial)

  ⇨ Resolve artificial dependencies through optimization

Superscalar Processor Phases

- Instruction fetching
- Instruction decoding
- Instruction issuing and parallel execution
- Committing state

Instruction Fetching

- Fetching multiple instructions each cycle.
- Challenge I: slow memory accesses
  - Separate instruction cache from data cache
  - High fetch speed to leave margin for cache misses

Instruction Fetching

- Fetching multiple instructions each cycle.
- Challenge II: branches
  - Recognize branch instruction (before real decoding)
  - Branch prediction and speculative execution
    - Static prediction using branch direction (backward branch forming a loop) and compiler flag
    - Dynamic prediction based on history
  - Compute branch target
    - Ideally not use a register (PC + offset), branch target buffer for repetitive loops
  - Transfer control
    - Delayed branches

Instruction Decoding

- Recognize and prepare instructions before (possibly parallel) execution
- Recognize data dependencies
  - Data in registers and main memory location
- Overcome artificial dependencies (WAR/WAW)
  - Register renaming
  - Larger physical register space or a temporary buffer

Register Renaming I: Mapping to More Physical Registers

- Model
  - Instructions specify logical registers (e.g., r1-r8). A larger number of physical registers in hardware (e.g., R1-R16).
  - Multiple versions of a logical register may physically exist at a time (in the case of WAR). Each version uses a physical register.

```
add r2, r1, 4
load r1, memaddr
```

```
add R2, R1, 4          mapping r1 ⇨ R1
load R3, memaddr       mapping r1 ⇨ R3
```

- Support on processor
  - Map logical to physical registers at instruction decoding.
  - Manage the resource of physical registers.
    - When can a physical register (e.g., R1) be reclaimed?

Register Renaming II: Using A Reorder Buffer

- Model
  - Results of instruction execution put in the buffer, arranged in instruction order
  - Results are committed in order

```
add r2, r1, 4

load r1, memaddr
```

```
add rob5, r1, 4        mapping r1 ⇨ r1
                       mapping r2 ⇨ rob5
load rob6, memaddr     mapping r1 ⇨ rob6
```


Parallel Instruction Execution

- Independent instructions can execute in parallel
- Subject to resource constraints
  - Physical register space or reorder buffer space
  - Function units (floating-point unit)
- Architectures
  - Queue-based, one queue per type of instructions (limited, cross-queue parallelism)
  - Reservation stations (Tomasulo's algorithm)

Committing State

- Committing execution results
  - Physical register becomes visible through logical register
  - Move result from reorder buffer head to the register
- Challenges of precise interrupts
  - Multiple instructions may commit in one cycle
  - Instructions may be committed out of order
- Possible solutions (operational state and recovery state)
  - Checkpoint and recover
  - Physical state (for operations) and architectural state (for recovery)

Handling Memory Operations

- Memory different from register
  - Target location requires calculation
  - Address itself a variable
  - Need logical to physical translation
  - ⇨ Only known at execution
- Consequences:
  - No recognition of dependency at decoding stage ⇨ no removal of artificial dependencies
  - Observe data dependencies during execution
    - Buffer outstanding access addresses; new load and store instructions checked against outstanding addresses

Handling Memory Operations

- Memory access requires logical to physical address translation
  - Accelerated by TLB (cached logical-physical mapping)
- Parallel memory access and address translation
  - Access cache before logical-to-physical address translation is done?

Role of Software

- Must run all legacy binaries correctly
- But software/compiler assistance can help things run faster
  - Software removal of dependencies
  - Place independent instructions together
  - Provide explicit hints for branch prediction
  - …