Tutorial 11

Final Exam Review

Introduction

- Instruction Set Architecture: contract between programmer and designers (e.g.: IA-32, IA-64, X86-64)
- Computer organization: describe the functional units, cache hierarchy (e.g.: opteron vs pentium 4)
- Computer architecture: manufacturing technology, packaging, etc (Pentium 4 vs Mobile Pentium 4)

Trends in technology

- <u>Bandwidth versus latency</u>: former increased much faster than the latter
- <u>Moore's law</u>: exponential growth in transistor count in ICs
- <u>Power consumption</u>: static (leakage) versus dynamic (switching) power consumption

Trends in IC cost

- ICs are produced on silicon wafers
- Multiple dies are produced per wafer
- Costs are split between:
  - <u>Fixed expenses</u>: masks
  - <u>Recurring expenses</u>: materials, manufacturing, testing, packaging, losses
- Wafers contain defects, and manufacturing can produce defective parts
  - <u>Yield</u>: proportion of good dies on a wafer

Trends in IC cost

- Four equations to determine the final cost of an integrated circuit
  - Dies per wafer
  - Die yield
  - Die cost
  - IC cost

$$ICCost = \frac{DieCost + DieTestCost + PackagingAndTestCost}{FinalYield}$$

$$DieCost = \frac{WaferCost}{DiesPerWafer \times DieYield}$$

$$DiesPerWafer = \frac{\pi \times WaferRadius^2}{DieArea} - \frac{\pi \times WaferDiameter}{\sqrt{2 \times DieArea}}$$

$$DieYield = WaferYield \times \left(1 + \frac{DefectDensity \times DieArea}{\alpha}\right)^{-\alpha}$$

Dependability

- Mean Time to Failure (MTTF): how long before a failure occurs on average
- Failures In Time (FIT): number of failures per billion hours (10^9/MTTF)
- Assume independent failures, exponentially distributed lifetimes

Locality

- A processor spends most of its time in small portions of code
  - Spatial locality: nearby addresses tend to be referenced together
  - Temporal locality: you reuse things you've accessed recently
  - Amdahl's law: compute the speedup resulting from an improvement in a certain portion of a system

$$Speedup_{overall} = \frac{ExecTime_{old}}{ExecTime_{new}} = \frac{1}{\left(1 - F\right) + \frac{F}{S}}$$

Performance

- CPU performance equation

$$\text{CPU\_Time} = \underbrace{\frac{\text{Instructions}}{\text{Program}}}_{\text{IC}} \times \underbrace{\frac{\text{ClockCycles}}{\text{Instructions}}}_{\text{CPI}} \times \underbrace{\frac{\text{Seconds}}{\text{ClockCycle}}}_{\text{CT}}$$

$$n = \frac{\text{ExecTime}_{\text{Y}}}{\text{ExecTime}_{\text{X}}}$$

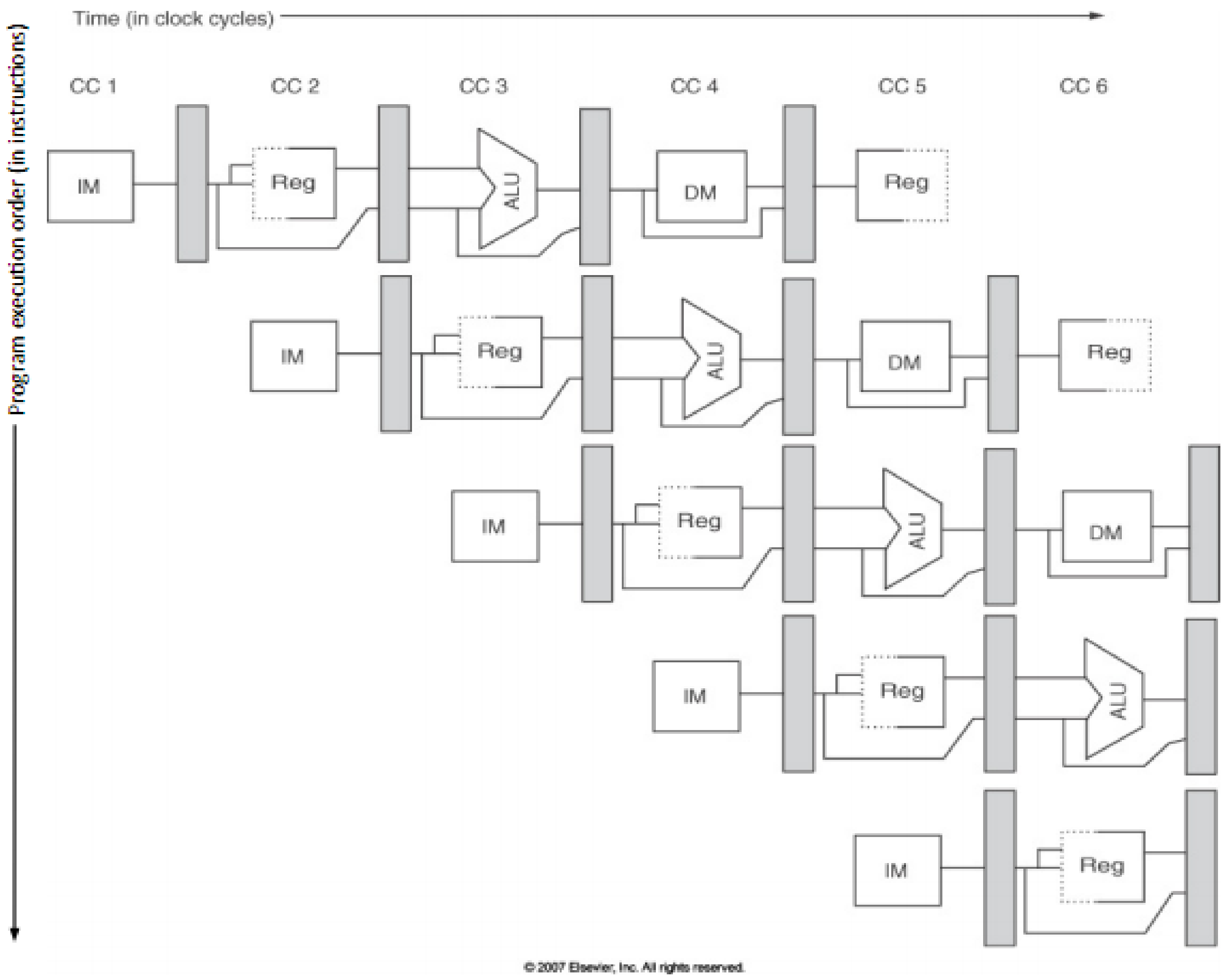
- Benchmarks: programs that allow you to get performance measurements by simulating real-life workloads
  - Geometric mean used to average unitless benchmark results, otherwise, use arithmetic mean

Pipelining

- Split an instruction in multiple consecutive stages which can be overlapped. Multiple instructions are in a different stage of execution at any given time
- N-stage pipeline means overlapped execution of N instructions. Ideal speedup=N.
- Never ideal!
  - New delays introduced by pipeline
  - Hazards: structural, data, control
  - Others: memory contention, pipeline imbalance

Pipelining

- Simple 5-stage RISC pipeline: IF/ID/EX/MEM/WB



Time (in clock cycles)
Program execution order (in instructions)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
IM
Reg
ALU
DM
Reg
© 2007 Elsevier, Inc. All rights reserved.

Pipeline hazards

- Structural: contention over a resource
- Data: unavailability of a result until a later time
  - RAW, WAR, WAW
  - RAW can be mitigated using forwarding
- Control: branch resolution causes a stall

- Many ways to mitigate hazards:
  - Compiler techniques: reordering, register naming, profiling-assisted branch prediction
  - Hardware techniques: speculation, register renaming, write buffers

Control hazards

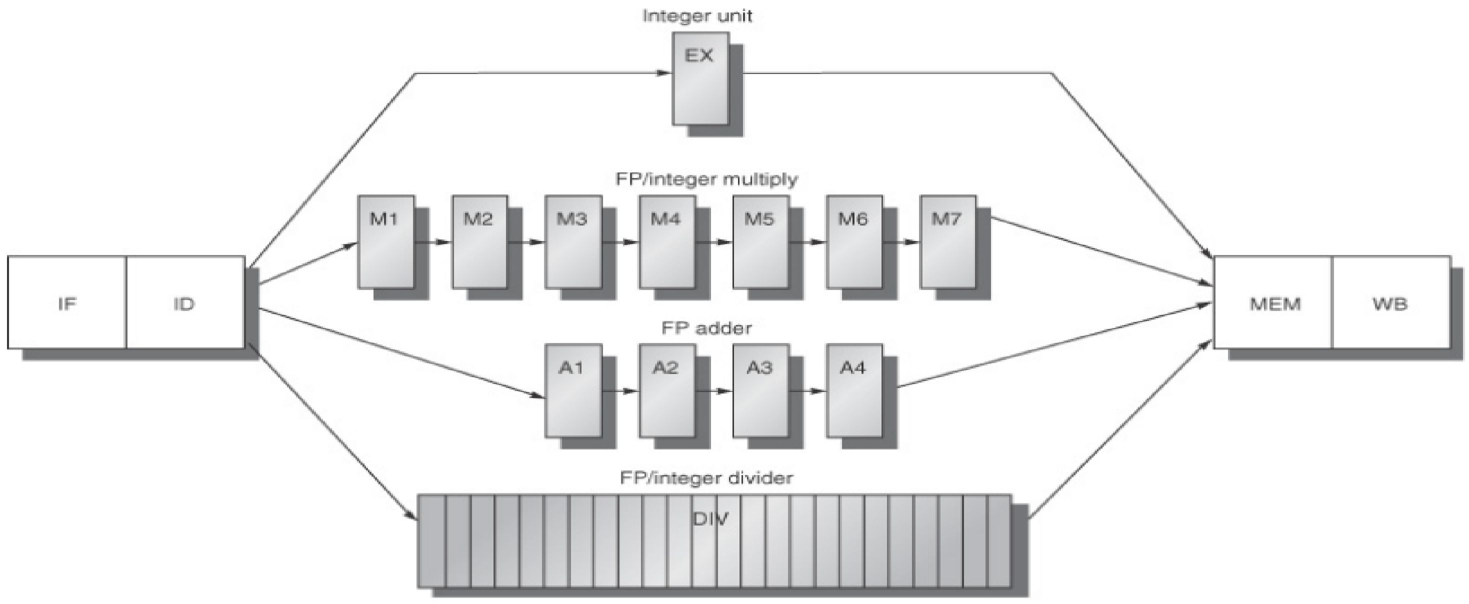
- Flush pipeline: when a branch is encountered, freeze for 1CC to resolve branch
- Predict not-taken: start executing PC+4
- Predict taken: start executing PC+offset (requires address resolution in IF)
- Delayed branch: insert “neutral” instructions right after a branch to give the CPU time to resolve the branch outcome

Exceptions

- Exceptional situations that disrupt a program
  - Arithmetic overflow, OS system call, div0, segfault, ...
- Various ways to qualify exceptions:
  - Synchronous vs asynchronous
  - User requested vs coerced
  - Maskable vs non-maskable
  - Within vs between instructions
  - Resume vs terminate
  - **Precise vs imprecise**

Multicycle operations

- The FP pipeline: separate pipelines for different types of operations: integer, FP mult, FP add, FP div
  - Those extra data paths either take more than 1CC or have multiple execution stages (add, mul, ...)



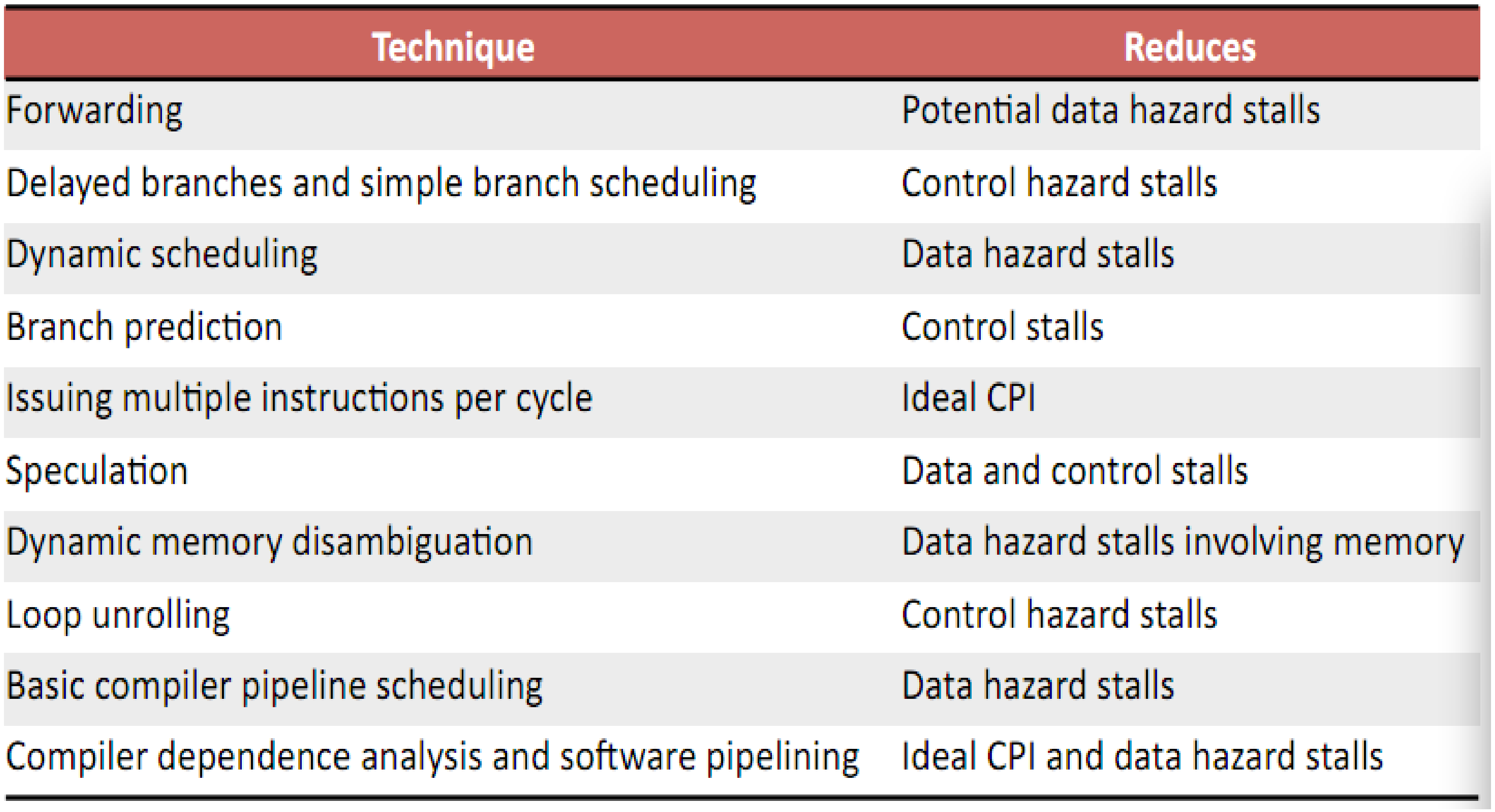
Multicycle operations

- Instruction can complete out of order: WAR, WAW hazards
- Exceptions can occur out of order
- New structural hazards: e.g.: DIV unit not pipelined
- Superpipelining: sub-dividing the operations further – e.g.: multicycle memory access

Instruction-level parallelism

- By overlapping the execution of multiple instructions, we obtain ILP
- To maximize ILP, we want to execute instructions in program order except when it doesn't affect the result of the program
- Three types of dependences which can cause hazards
  - <u>True/data</u>: an instruction depends on a result produced by a previous instruction (RAW)
  - <u>Name/anti</u>: two instructions use the same memory location, but don't exchange information (WAR)
  - <u>Name/output</u>: two instructions write to the same memory location, and a third reads it before the second has properly written to it (WAW)
  - <u>Control</u>: dependence on the outcome of a branch

Instruction-level parallelism



Technique	Reduces
Forwarding	Potential data hazard stalls
Delayed branches and simple branch scheduling	Control hazard stalls
Dynamic scheduling	Data hazard stalls
Branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Speculation	Data and control stalls
Dynamic memory disambiguation	Data hazard stalls involving memory
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Compiler dependence analysis and software pipelining	Ideal CPI and data hazard stalls

Loop unrolling

- We want to keep the pipeline full
- Replicate the body of a loop multiple times to find ILP and reduce the number of control dependences
- This technique yields larger executables, requires lots of registers
- We don't always know if we can unroll a loop since the upper bound is not always static

Branch prediction

- No prediction: flushing, delayed branch
- Static prediction: predict taken, not-taken
- Dynamic prediction: use past behavior
  - 1-bit predictor: repeat past outcome
  - 2-bit predictor: repeat past outcome provided it has occurred at least twice in a row
  - Correlating predictor: (m,n) predictor uses 2^m n-bit predictors, depending on the outcome of the past m branches
  - Tournament predictor: two predictor – one local, one global. Dynamically select between the two

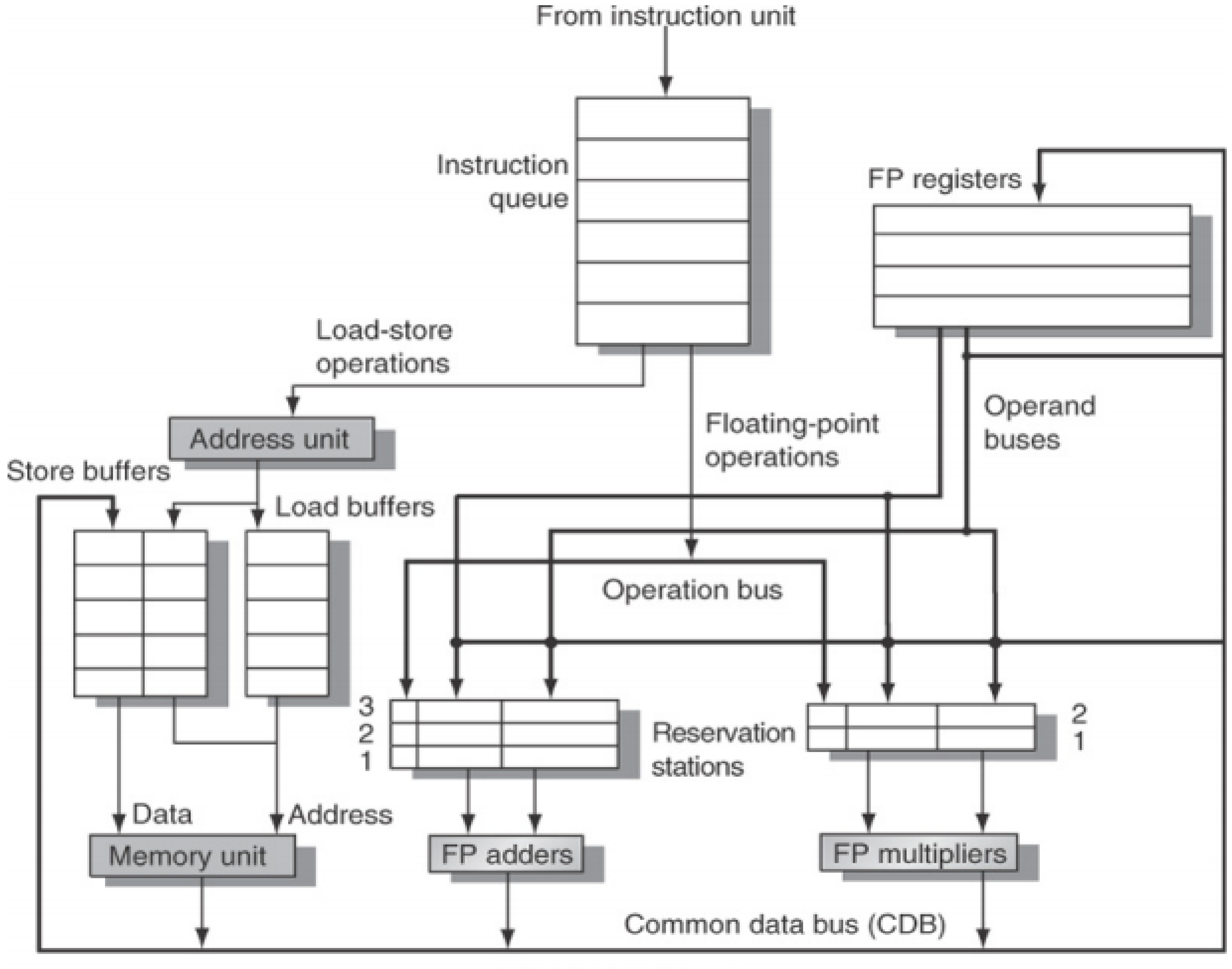
Dynamic scheduling

- Hardware rearranges instructions to reduce stalls
  - Allows out-of-order execution and completion
  - Two stages instead of ID:
    - Dispatch: decode and check for structural hazards (RS, ROB)
    - Issue: wait on data and structural hazards (FU)
  - In-order dispatch, but instructions can issue out-of-order and execute out-of-order
  - Tomasulo's algorithm for FP operations

Tomasulo's algorithm

- Performs implicit register renaming to get rid of WAR and WAW hazards
- Uses reservation stations to wait on RAW hazards
- Use a common data bus to broadcast and listen for execution results
- Two-step loads and stores: compute effective address, put in load/store buffers
- Can be adapted for speculative operation

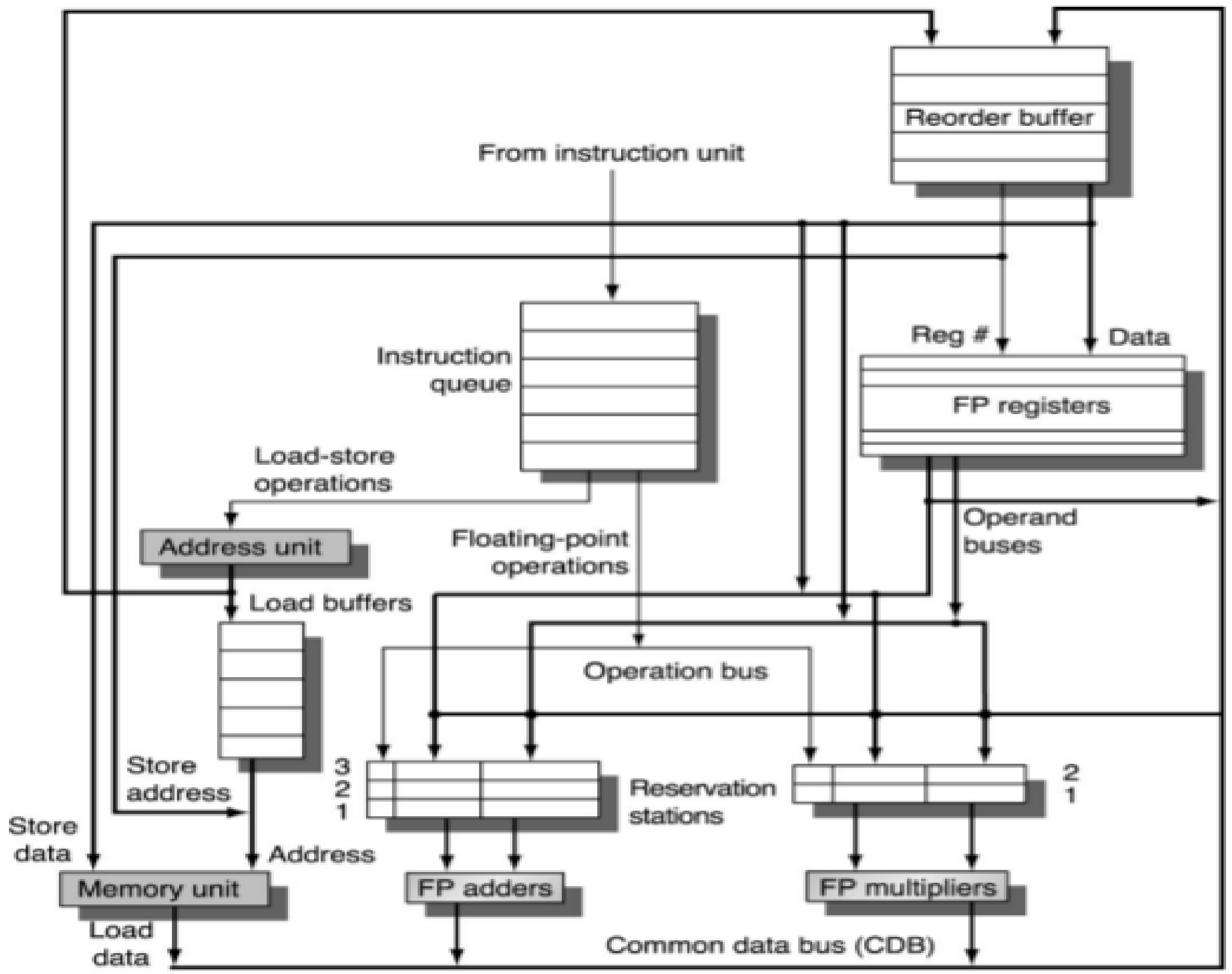
Tomasulo's algorithm



Speculative Tomasulo

- Branching can limit the ILP exploitable by Tomasulo's algorithm
- Speculate on branch outcome and start executing the instructions that follow the branch
- Dangerous: can modify the processor state irreversibly or raise unwanted exceptions
  - Keep track of speculative execution and undo those
  - Execute o-o-o but commit in-order
  - Use re-order buffer to hold uncommitted results
  - Register file updated only when instructions commit
  - RS buffer instructions between issue and execution, but renaming done by ROB
  - Mispredicted branches flush the later ROB entries and restart execution
  - ROB can provide precise exceptions since it commits in order

Speculative Tomasulo



Multiple-issue processors

- A CPU can have parallel pipelines
- Superscalar: schedule instructions when possible. More than one can issue at once.
  - Static: fetch multiple instructions at once, issue all if possible. In-order execution.
  - Dynamic: fetch multiple instructions at once, dynamic scheduling thereafter = o-o-o execution
- VLIW: (static superscalar, no hw hazard detection) pack instructions into fixed-size long words. Statically scheduled by the compiler

Memory hierarchy

- Memory performance has not increased as quickly as processor performance
- We would like an unlimited amount of very fast memory, but it's not feasible
- Principle of locality: temporal/spatial
- Hierarchy of memories, progressively larger but slower, organized in levels
- Multiple levels of cache down to main memory
  - Hit/miss in a cache

Memory hierarchy

- <u>Memory accesses per instruction</u>: to fetch the instruction and (possibly) load/store to memory
- <u>Miss penalty</u>: how many CC to get the data from a slower memory
- <u>Miss rate</u>: how often you miss in a cache

$$\begin{aligned}\text{MemoryStallCycles} &= \text{NumberOfMisses} \times \text{MissPenalty} \\ &= \text{IC} \times \frac{\text{Misses}}{\text{Instructions}} \times \text{MissPenalty} \\ &= \text{IC} \times \frac{\text{MemoryAccesses}}{\text{Instruction}} \times \text{MissRate} \times \text{MissPenalty}\end{aligned}$$

Caches

- <u>Block placement</u>: where to do you put a block? (direct mapped, set associative, fully associative)
- <u>Block identification</u>: how do you know you have the right block? (tag, index, offsets)
- <u>Block replacement</u>: what do you do when a spot is taken in the cache? (Random, LRU, FIFO)
- <u>Write strategy</u>: what do you do on a write? (write through, write back)
- <u>Write miss strategy</u>: what do you do on a write miss? (write allocate, no-write allocate)

Caches

- How cache speedups can be calculated, can be integrated in the CPU time equation
- Cache addressing
  - Index
  - Tag
  - Offsets
- Validity bit, dirty bit
- The three Cs of cache misses
  - Compulsory: you've never accessed this data before
  - Capacity: you need too much information
  - Conflict: mapping rules map too many blocks to the same location

Caches

- Miss rate
- Multi-level caches
  - Global miss rate
  - Local miss rate
- Unified vs split caches (data vs instructions)
- Average Memory Access Time (AMAT)

$$\begin{aligned}\mathrm{CPUTime} &= \mathrm{IC} \times \left(\mathrm{CPI}_{\mathrm{base}} + \frac{\mathrm{MemoryStalls}}{\mathrm{Instruction}}\right) \times \mathrm{CCTime} \\ &= \mathrm{IC} \times \left(\mathrm{CPI}_{\mathrm{base}} + \frac{\mathrm{MemoryAccesses}}{\mathrm{Instruction}} \times \mathrm{MissRate} \times \mathrm{MissPenalty}\right) \times \mathrm{CCTime}\end{aligned}$$

Caches optimizations

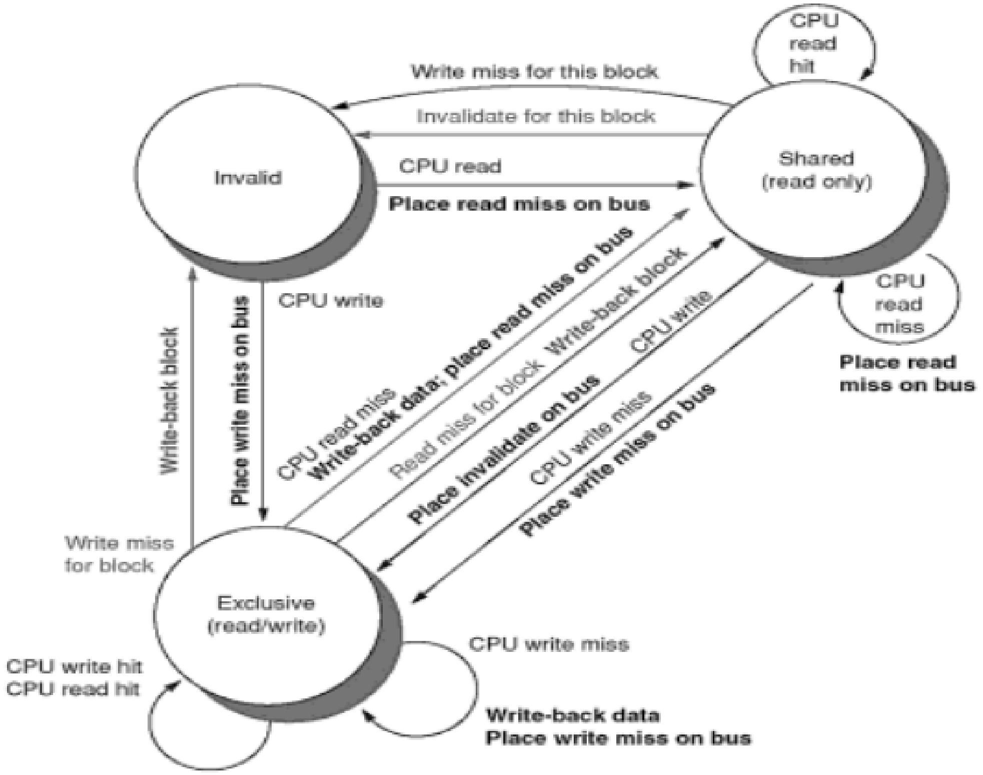
- Reduce the miss rate
  - Increase block size
  - Increase cache size
  - Increase associativity
- Reduce the miss penalty
  - Multilevel caching
  - Prioritize reads over writes
- Reduce hit time
  - Avoid address translation during cache indexing

Cache coherence protocols

- Coherence: memory accesses to a single location are seen by all CPUs in order
- Consistency: memory accesses to various locations are seen by all CPUs in order
- Two classes of protocols:
  - Directory based: central location
  - Snooping: global bus
- Two ways to ensure cache coherence:
  - Write invalidate: broadcast invalidation messages
  - Write update: broadcast new value

Cache coherence protocols

- Snooping + write-back + write-invalidate
- Three states: Invalid, modified, shared



Virtual memory

- Each program gets a very large virtual address space which is then mapped to memory.
- Move pages in and out of memory as you need them, depending on RAM size
- Good for multitasking: isolation, memory sharing because virtual memory is relocatable
- The sum of all the memory needed by all programs can be larger than RAM. Use disk as extra memory space.

Virtual memory

- Virtual memory requires a mapping to physical memory
- Block placement: Fully associative
- Block identification: page table
- Block replacement: LRU to minimize page faults
- Write strategy: write-back because disk is slow
- Translation look-aside buffer (TLB) used to store recent translations

Final exam

- December 9th at 9am
- Allowed 2 double-sized crib sheets
  - You have to hand those in with your exam
- 3h exam
- 120 points total
  - 5 problems: 20pts each = 100pts
  - 20 short answer questions = 20pts