ECE552: Computer Architecture

Arnav Patil

University of Toronto

Contents

1 Design Goals

- Functional – needs to be correct for what function it supports
- Reliable – does it perform correctly consistently?
- High performance – “fast” is only meaningful in context of a set of important tasks
- Low cost – per unit manufacturing cost (wafer) and cost of making the first chip (mask) and design cost (huge teams of engineers)
- Low power – energy in and energy out
- Secure – can our design protect important info?

Aside on Security

- Architecture – timing-independent functional behaviour of a computer
- Microarch – implementation techniques to improve performance
- Meltdown – leaks OS kernel memory, fixed
- Spectre – leaks memory outside-of-bounds checks and sandboxes.
- ECE552 focusses on microarch techniques for performance
  - Must understand the ramifications of these technologies

Microprocessor Revolution

- 1980: enough transistors (50k) to put full processor on a single chip
  - Fewer inter-chip crossings
  - Only 1 mask required now
- Microprocessors created new market segments
- Replaced incumbents in existing segments
- How are growing transistor counts utilized?
  - Implicit parallelism
    - Extracting implicit instruction-level parallelism (ILP) e.g. pipelining and caching
  - Deeper pipelining (5-stage pipeline), branch prediction, multiple issue, superscalar
  - Dynamic scheduling – out-of-order execution
  - Explicit parallelism – supports data- and thread-level parallelism
  - Coherent caches and synchronization primitives

Application Pull

- Corollary to Moore’s Law – cost halves every 2 years
- Computers are cost-effective for: national defence, enterprise, dept computing, pervasive computing

Application-Specific Chips

- Course mainly focusses on general-purpose CPUs
- Large, profitable, segment of application-specific CPUs

Layers of Abstraction

- Only way of dealing w/ complex systems
  - ISA, microarch, systems arch, tech, applications

Instruction Set Architecture

- HW/SW interface, supports OS functions, is a good compiler target
- HW impacts – efficient and parallel implementations
- Good ISA entails abstraction w/o interpretation

Microarchitecture

- RTL design
- Implement instruction set and explicit capabilities
- Iterative process
- Delivering sequential and parallel application
  - Deep pipelining, multiple issue
  - Dynamic scheduling
  - Branch speculation and multi-core

2 Performance

- Must discuss metric to evaluate modern architectures and Moore's Law effects
  - Moore's Law slowing but had a historical trend

Empirical Evaluation

- Metrics – performance, cost, power, reliability
- Metrics more important in combination than individual
  - Performance per cost (MIPS/$)
  - Performance per power (MIPS/W)
- Basis for design and purchasing decisions

Performance

- Metrics are latency and throughput
- Reporting performance – benchmarking and averaging
- CPU performance equation
- Latency (execution time) – time required to finish one fixed task
- Throughput (bandwidth) – number of tasks completed in a fixed time
- Often contradictory – throughput can exploit parallelism but latency cannot
- Choose definition that matches goals (usually this is throughput)

Performance Improvement

- Processor A is X times faster than B if
  - Latency(P,A) = Latency(P,B)/X
  - Throughput(P,A) = X·Latency(P,B)
- Processor A is X% faster than B if
  - Latency(P,A) = Latency(P,B)/(1+X/100)
  - Throughput(P,A) = Throughput(P,B)·(1+X/100)
- What is P in Latency(P,A)?
  - Processor executes some program, but which?
- Actual target load – accurate, but not profitable, repeatable, overly specific
- Some representative benchmark program
  - Portable and repeatable, accurate if benchmark matches intended workload
  - But hard to pinpoint problems, may not be exactly what you want to run
- Some small microbenchmarks
  - Portable and repeatable, easier to run
  - Easy to pinpoint problems
  - Downside is that microbenchmarks are not representative of the programs you might actually want to run

Adding/Averaging Performance Metrics

- We can add latencies for P1 and P2 but not throughputs

$$\text{Throughput}(P1+P2,A) = \frac{2}{\frac{1}{\text{Throughput(P1,A)}} + \frac{1}{\text{Throughput(P2,A)}}}$$

- Same goes for means
  - Arithmetic

$$\frac{1}{N}\sum_{1}^{N}\text{Latency}(P)$$

  - Harmonic

$$\frac{N}{\sum_{1}^{N}\text{Throughput}(P)}$$

  - Geometric

$$\sqrt[N]{\Pi_1^N \text{Speedup(P)}}$$

Iron Law of Performance

- Multiple aspects of performance, helps to isolate them
- Latency(P,A) = S/prog

$$\frac{\text{Instruction}}{\text{Program}} \times \frac{\text{Cycle}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

- Instructions/program – dynamic instruction count
  - Function of program, compilers, ISA
- Cycles/instruction – CPI
  - Function of program, compilers, ISA, microarch
- S/cycle – clock period
  - Function of microarch and tech parameters
- For low latency, minimize all three, but they pull against one another

Measuring CPI

- Execution time, CPI
- CPI breakdowns – hardware event counters, cycle-level microarch simulation

Improving CPI

- Course focusses on CPI improvements more so than clock frequency
  - Historically clock accounts for 70% of performance
- Now things have changed
  - Deep pipelining no longer power-efficient
- Some techniques – caching, speculation, multiprocessing
- Amdahl's Law – you don't want to speed up a small fraction of a program to the detriment of the rest
  - f – fraction that can be parallely speedup
  - $1 - f$ – fraction that must execute serially

$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{N}}$$

$$\lim_{n\to\infty} = \frac{1}{1-f}$$

- Pretty good ideal scaling for a modest number of cores
- Large number of cores requires a lot of parallelism

3 Pipelining

Sequential Model

- Implicit model of all modern commercial ISA
- Called the von Neumann model, but was implemented in ENIAC beforehand
- Basic feature, the program counter
  - Defines total order on dynamic instruction
  - Next PC = PC++ unless instruction says so
  - Order and named storage defines computation
    - Value flows from X to Y via storage element A iff X names A as an output, and Y names A as an input, and X appears before Y in the total order
- Processor logically executes loop
  - Instruction execution assumed to be atomic
  - Alternatives have been proposed
- Datapath – functional units (ALU), registers, memory, interface (data cache)
- Cache (decode portion) – muxes, write enables, etc
  - Regulate the flow of data in the datapath
  - Translates ops into control signals

Breaking Down Instructions

- Instruction Fetch (F)
  - Instruction fetched from memory at adder pointed to by PC, then PC does PC++
- Instruction Decode (D)
  - Decode instruction to find type
- Execution (X)
  - ALU, compute memory addresses, branch update
- Memory Access (M)
  - Load and store
- Writeback (W)
  - Write instruction result back to registers
- Single-cycle control is hardwired
  - Low CPI (exactly 1) but long clock period
  - Has to accommodate for the slowest instruction
- Multi-cycle control – micro-programmed/state machine
  - Lower clock period but higher CPI
- When breaking into stages – can't divide evenly

  - 11, if not 10 (50/5)
  - Due to uneven combinational paths and increased latency through latches

- Latency vs Throughput
  - Latency – no good way to make a single instruction go faster
  - Throughput – fortunately, we do not care about single instruction latency

- Goal: make whole programs go faster

Pipelining

- Improves instruction throughput over instruction latency
- Begin with the multi-cycle design
  - When instruction advances from stage 1 to 2, allow the next instruction to enter stage 1
  - Instruction-stage parallelism
  - Assume all instructions take some number of stages
- Point is that instructions will enter and exit the processor at a faster rate
  - Multiple instructions in-flight simultaneously
- Some values like PC, IR have to latched at every stage as multiple instructions in-flight
- Pipeline registers named by stages they separate

Pipeline Control

- One single-cycle controller, but also pipeline the control signals

Terminology and Foreshadowing

- Scalar pipelines – 1 instruction per stage per cycle
- Alternative is superscalar – multiple per stage per clock cycle
- In-order – instructions enter execute stage in program order, alternative is out-of-order

Pipelining Goals

- Balanced – all stages should take roughly the same time
- Doesn't make sense to optimize any stage that's not the longest
- For in-order pipelining, instruction must pass through all stages and in the same order
- Buffering – not all stages take some amount of time
- Independent computations
  - No relationship between work units
  - Minimize pipeline stalls

Pipeline Performance Calculations

- Why is pipeline CPI > 1 ?
  - CPI for scalar in-order pipelines is 1+ penalties
  - Stalls used to resolve penalties for hazards
    - Hazard: condition that jeopardizes sequential illusion
    - Stall: pipeline delay introduced by hardware to restore the sequential illusion
- Long penalties are okay if they happen very rarely
- Stall also have implications for ideal number of pipeline stages

Managing a Pipeline

- Proper flow requires pipeline operations
- Operation I: stall
  - Effect – stop instruction at current stage
  - Usage – make younger instruction wait for other instructions to complete
  - Implement – de-assert write enables for pipeline registers
- Operation II: flush
  - Effect – remove instruction from current step
  - Usage – control (later) so they never appeared
  - Implement – assert clear signals and replace with NOOPs

Dependences and Hazards

- Dependence – relationship that serializes two instructions
  - Data: 2 instructions use the same value of named storage
  - Control: 1 instruction controls when another executes
- Hazards – dependence causes potential incorrect execution
  - Possibility of using wrong data/corrupting it
  - Can be fixed with stalls for multiple stages

Data Hazards and Dependences

- Three types of data dependences
  - Read-after-write (RAW)
  - True dependence – value flows through its true dependence, using different output register for add doesn't help

RAW: Detect and Stall

- Stall logic in decode stage to detect and stall reader
- Same thing for second source register
- Reevaluate every cycle until no longer true
- Pro: low-cost, simple
- Con: IPC degradation, RAW dependencies are common
- Main thing it does is deassert write-enable signals
  - Also want to stop some values from propagating so we replace w/ NOOPs, called pipeline bubble

Reducing RAW Stalls With Bypassing

- Why wait until W? Data available after X or M
- Bypass or forward data directly to input of X or M stage
- MX: from beginning of memory to input of X
- WX: from start of W to input of X
- WM: from start of W to input of M
- Bypass logic similar to but separate from stall logic
  - Complement one another, can't bypass, must stall
  - Stall logic controls latches, bypass logic controls muxes

Write-After-Write Hazard/Dependence

- Compiler effects – scheduling problem, reordering would leave wrong values in registers
- Artificial – no value flows through dependence
- Eliminate by using different output registers
- Pipeline effects – doesn't affect in-order pipeline w/ single-cycle execute operation
- Could have WAW hazard with multicycle

Write-After-Read Hazard/Dependence

- Compiler effects – scheduling problem, reordering
- Artificial – solve by using a different output register name
- Pipeline effects – hazard can't happen in simple in-order pipeline
  - BUT could happen in an out-of-order execution

Data Hazards Summary

- Real instructions have no read-after-read dependence
- RAW – true dependence
- WAR – anti-dependence
- WAW – output-dependence
- Data hazards are a function of data dependencies and the pipeline
  - Potential for executing dependent instructions in the wrong order
  - Requires both instructions to be in-flight

Pipelined Functional Units

- Let's first look at decomposing execute stage
  - Fast integer arithmetic and logic operations, 1 cycle
  - Slow integer arithmetic and logic operations, 2 cycles
  - Floating point, add, multiply, divide take multiple cycles
- How many stages depends on many factors
- When we have "long" operations – RAW stalls become more frequent
- WAW hazards now possible if we're naive about the design
- WAR hazards are not possible anymore as register values always read in decode

Structural Hazards

- When measures are required twice in the same cycle
- Or when instructions and data cache share some structure
- Pro: low cost, simple fix
- Con: increases CPI
- To fix, we use a proper ISA and pipelined design
- Best to avoid by design, each instruction uses each named structure exactly once for at most 1 cycle and always in order

Control Hazards

- Pipeline works well when there is no transfer of control
- F always gets the next instruction, but there is a problem is the sequential flow is disrupted

4 Control Speculation

- How to fix control hazards caused by pipeline?
  - Option I: always stall at decode once we see it's a branch
  - Option II: assume all branches are not taken, do PC=PC+4, then recover if wrong

Control Speculation and Control Recovery

- Speculation – predict branch outcome and recover if the guess is wrong
- Misspeculation recovery – what to do on the wrong guess
- Branch resolves in X – younger instructions in F/D haven't changed permanent state, so we can flush F/D and D/X registers

Branch Prediction

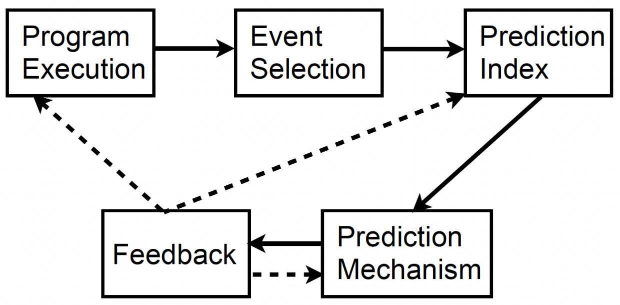
- Taken branches more common in loops
- Yield taken/not taken prediction w/ high probability of being right

Big Idea: Speculation

- Speculation – engage in risky transaction on a chance of profit
- Speculative execution – execute before all parameters are known
- Correct – avoid stall, improve performance
- Incorrect – flush/abort incorrect instruction, must undo changes and RECOVER pre-speculative state
- The game: $[\%_{correct} \times gain] > [(1 - \%_{incorrect}) \times penalty]$
- Unknown parameter – are the instructions to execute next correct or not?
- Mechanics –
  - Guess branch target, start fetching at guessed position
  - Execute branch to verify guess
  - Don't write to register or memory until prediction is verified

Dynamic Branch Prediction

- BP Part I – direction predictor
  - Applies to conditional branches only, which are hard to predict
  - Predict taken or not taken
- BP Part II – target predictor
  - Applies to all control transfers
  - Supplies target PC
  - Done in F stage (or earliest possible in a deep pipeline)
    - Determines if instruction is control prior to D stage
  - Easy to implement
- Predict in F
  - Don't know if an instruction is a branch but we can disregard our prediction if it's not
- Direction predictor (DIRP)
  - Map conditional-brnach PCs to taken/not-taken decisions



  - Difficult to do well
  - Individual branches are either weakly-biased or unbiased

- Branch Prediction Buffer (BPB)
  - 1-bit predictor, PC indexes table of bits, no tags
  - Essentially branch will go same way it went the last time
- We see that the 1-bit predictor changes its mind too easily so we use a two-bit saturating counter instead
- Force DIRP to mispredict twice before changing state

Storing and Addressing Predictions

- Think of it like memory/cache, but we want to avoid tags
- Aliasing could happen if 2+ instructions' PC accesses the same counter – also under-utilization
  - But this is OKAY because we're only making a prediction, the (small) penalty we pay is in performance

Correlated Branch Behaviour

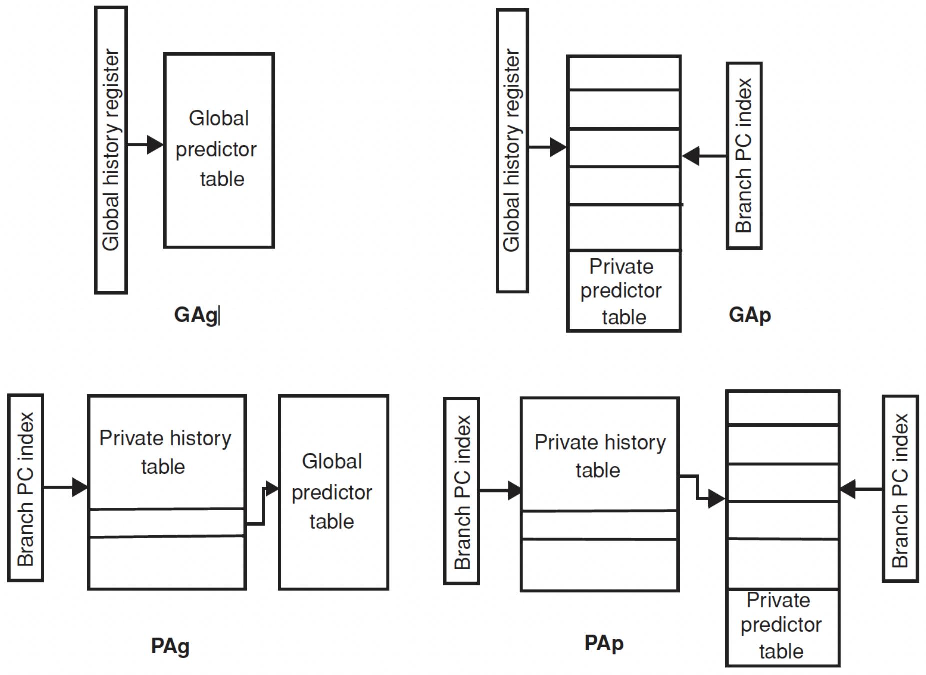
- 2-bit scheme – small amount of history (local scheme)
- History register – shift in outcome every branch
- Record last k branch outcomes
- Correlated (2-level) predictors
  - Exploits observation that branch outcomes are conditional
  - Maintains separate prediction per PC **or** BHR

Correlated Predictors

- Design Choice I – one global BHR or one per PC
  - Each captures different kind of behaviour
  - Global often better – captures local patterns for tight loops
- Design Choice II – how many history bits (BHR size)?
  - Tricky question to answer
  - Long BHRs are good for some, shorter ones are better for others

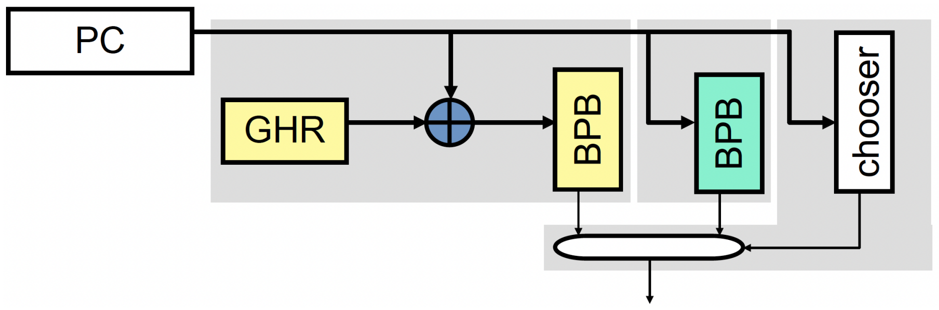
  - BPB utilization decreased w/ longer BHR – many history patterns end up never seen
- Using PC and BHR allows multiple PCs to dynamically share BPBs
- Long BHRs demand longer training

Two-Level Predictors



Hybrid (Tournament) Predictors

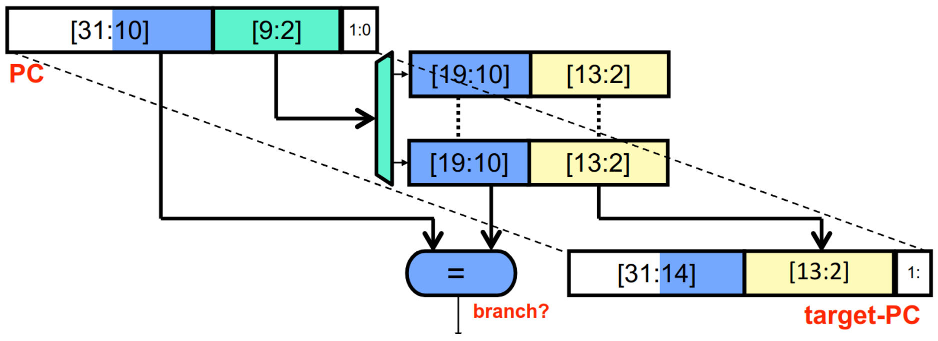
- Attacks correlated predictor BHT utilization pattern
- Idea: combine 2 predictors
  - Simple BPB predicts history independent
  - Correlated predictor predicts only branches that need history
  - Chooser assigns branches to one predictor over the other
  - Branches start in simple BPB, more mis-speculation threshold



- Lots of direction prediction strategies
  - Perceptron based predictions (used in latest AMD processors)
  - Further exploration/research in Lab 2
- Also need to predict branch targets

Branch Target Buffer (BTB)

- Need to predict branch target
- Small cache:



- If the cache hits: this is a control instruction and its going to the target PC (if taken)
- If the cache miss: not a control instruction OR never seen before OR evicted
- Why are partial tags ok? Full tags not necessary
  - Target PC is just a guess – aliasing is OK

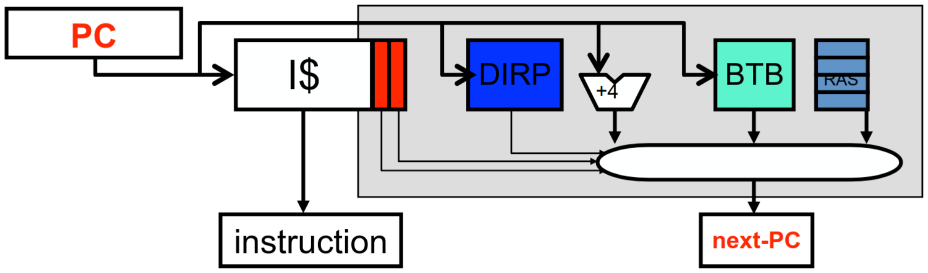
Why Does a BTB Work?

- Control instructions' targets are stable
- Direct means constant target – indirect means registers target
- More on indirect calls – so MIXED BAG
  - (dynamically linked) – yes
  - (dynamically dispatched or virtual) – no
- Indirect conditional jumps (conditional statements) – no
- Return instruction? Indirect, so NO, but
- Polymorphism and virtual functions can kill performance

Return Address Stack (RAS)

- Return addresses are easy to predict w/o a BTB
  - Return addresses stack call sequence
    - Call – push PC+4 into RAS
    - Predict for return is RAS[TOS]
- How do we tell if an instruction is a return before decoding?

- Attach pre-decode bits to I$
  - Written after 1 time the instruction executes
  - Useful bits? return



- Importance of accuracy increase with depth and width of pipeline
- Basic building block – 2-bit saturating counter
- Predict prediction and target

How Are Interrupts/Exceptions Handled?

- Interrupt – external requires control of the processor, e.g. timer, I/O device req
- Exception – internal events so rare that we don't handle in hardware so we defer in software e.g. division by 0, page fault, memory protection, violation, illegal instruction
- Can occur at any stage except W
  - Page fault – F or M
  - Illegal op code – D
  - Divide by 0 – X
- Upon detecting an interrupt
  - OS saves processor state
  - Interrupt handling routine
    - Abort program
    - Corrupt program

Handling Interrupts/Exceptions

- Instructions before `i`, (`i-1`, `i-2`) are currently in pipeline are completed normally
- Results get saved
- `i` and instructions after get flushed, converted into NOOPs
- Saved PC is PC of `i`
- Called precise state/interrupt

Handling Precise Exceptions

- Flag inserted into pipeline register
- Instruction converted into NOOP
- Instruction handled in M/W register as exceptions must be handled in program order and NOT temporal order
- What about 2 in-flight exception?
  - Must defer handling interrupts until writeback and force in-order writeback
  - Bottom-line: maintaining precise state is important but hard

5 Dynamic Scheduling

- Problem with in-order scheduling – instructions w/ no dependence can't advance b/c of dependent instructions are in its way

Static Instruction Scheduling

- Issue: time at which instruction begins its execute stage
- Schedule: order in which instruction execute
- Scheduling: actively rearranging instructions to enable rapid issue
- Static scheduling done by compiler – knows pipeline and program dependencies
- Control flow messes with rearranging instructions
- E.g. loop unrolling – increases scope for loops

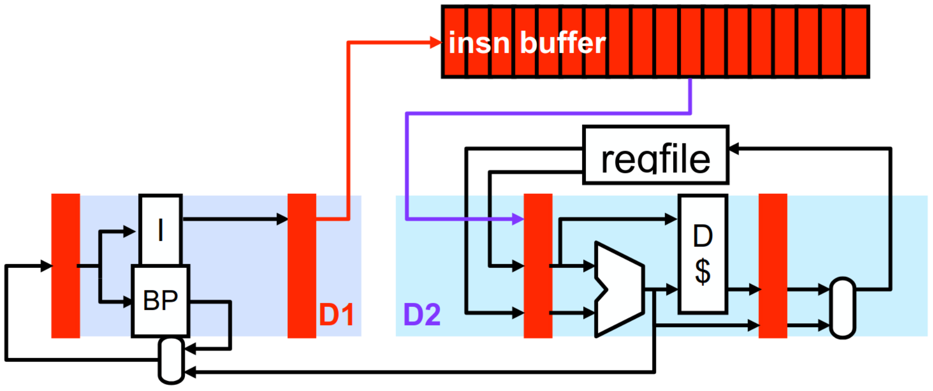
Motivation: Dynamic Scheduling

- AKA out of order execution – break von Neumann order of instruction execution
  - Pros: reduce RAW stalls, increase pipeline and functional unit utilization
  - Originally for FP utilization
  - Expose more opportunities for parallel issue
- BUT it has to look like we never broke the illusion of sequential order

Before We Continue...

- If it can be done in software, why implement in HW?
  - Performance portability – don't want to recompile for new machines
  - More information available – memory addresses, branch directions, cache misses
  - More registers available – compilers might not have enough to fix WAR and WAW hazards
- Easier to speculate and recover from a mis-speculation

Instruction Buffer



- Trick: instruction buffer – conceptually a bunch of latches holding instructions
  - Gives us our scheduling scope
  - Split Decode (D) into 2:
    - ∗ D1: accumulate decoded instructions into buffer in order
    - ∗ D2: buffer should send instructions out-of-order to rest of the pipeline

Scheduling Algorithm I: Tomasulo

- Tomasulo's algorithm – removes WAR/WAW hazards
- Reservations station (RS) – instruction buffer
- Common data bus (CDB) – broadcasts results to RS
- First implementation: IBM 360/91
  - Initially only for floating point ops
- We will implement simple – dynamic scheduling for all instruction types

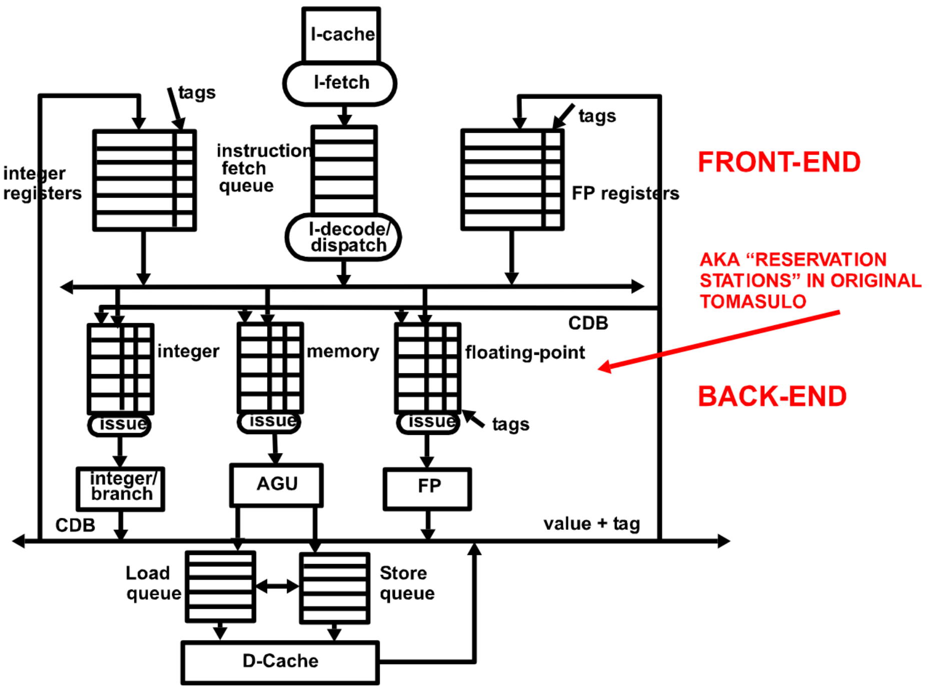
Name Dependence

- 2 instructions use the same register but no data flow
- Anti-dependence – WAR
- Output-dependence – WAW

Register Renaming – In Hardware

- Change register names to eliminate WAR/WAW
- Elegant idea (cache/pipeline)
- Think of registers as names, not physical storage locations
  - Can have more locations than names
  - Multiple active versions of the same register names
- Map-Table: maps names to most recent locations

- SRAM indexed by name
- On a write – allocate a new location, note mapping in the table
- On a read – find storage location of most recent version via map-table lookup
- Detail – must deallocate locations once we're done
- Renaming should remove WAR/WAW dependences and leave RAW dependences intact



Tomasulo Data Structures

- Reservation station (RS)
  - stores FU, busy, op
  - V_j, V_k – source register value, captured when entering the RS
  - Q_j, Q_k – source register tags (RS# of RS that will eventually produce that support)
- Map-table – implemented as RS# that will write into that register
- Common Data Bus – broadcasts <`RS#, value`>
- Tags implemented as ready-bits
  - Tag = 0 – value is ready
  - Tag != 0 – value is not ready, watch for CBD broadcasts of that tag

New Pipeline – F D S X W

- Fetch – same as before
- Dispatch
  - Stalls for structural hazard – is the target RS for that instruction full?
  - Input data ready? If tag bits are 0, we can read the value into the RS
    - Else, if not ready (tag != 0), can read value into the RS
  - Set register status (update Map Table) – rename output reg to new physical location RS#
- Issue
  - Wait for RAW hazards and structural hazards
  - If not busy and all inputs are ready, will read values from RS and send them to execute
- Execute
  - Combined w/ memory, do the same as before
- Writeback
  - Wait for structural hazard (CDB)
  - If CBD available, output register
    - Does output register status still match (tag status)?
    - If yes, clear the tag and write value to register file
  - Do any instructions in any RS care about this result?
    - If they match once broadcast on CBD, clear tag and copy value
- Remember, this can all happen simultaneously, not mutually exclusive
- Wait vs. stall, stall propagates to younger issues, but a wait only holds up that instruction and doesn't affect younger ones behind it

Value/Copy-Based Register Renaming

- Tomasulo-based register-renaming
  - Called "value-based" or "copy-based"
  - Names – architectural registers
  - Storage locations – register files and RS's
  - Values can (& do) exist in both
- Register files hold most recent values
- RS versioning eliminate WAR hazards
- Storage locations referred to internally by RS# tags
  - Map table translates names to tags
  - Tag = 0, value is in the register file
  - Tag != 0, value is not ready and is being computed by RS#
  - CDB broadcasts value w/ tags attached so instructions know what value they are looking for
- This means we can have multiple loads outstanding

- Issue: wait for RAW hazards, and if none, then will read values from entry and sent to functional unit
- If an instruction needs to read the register file in the same cycle a value is written back to the regfile & RS, we can assume internal forwarding
- Similar to W-D internal forwarding in the basic pipeline
- Check values being written back on CDB first, then instruction and dispatch in the second half of the cycle
- Promise kept! In-order dispatch, out-of-order execution and completion

Tomasulo Summary

- Instruction buffer/RS – stall in D – structural hazard
- Functional unit – wait in S – structural
- RAW – wait in S
- WAR – none – fixed by renaming, same goes for WAW

Dynamic Scheduling as Loop Unrolling

- Combine iteration – implement `i` and `i+1`, increment by `i+2`, more flexible
- Pipeline schedule – reduce impact of RAW hazards
- Rename registers – rename to fix any WAR/WAW hazards, may come from reordering
- Renamings set up a data flow graph
- Say an instruction is writing to a nonzero tag in the register file
  - Will simply overwrite tag, result of first load will never be written to reg file
  - Architecturally this is OKAY!!

Branches

- 2 options wrt how branches are dealt with
  - No speculation – branch must be resolved before younger instructions can writeback
  - Must have a way to recover from a mis-speculation
- An assumption – we can issue the same cycle as we see a tag match on the CDB
- Say both multiply RSs are used, then we can't add the 3rd multiply to RS
  - Structural hazard that results in a stall
  - Stall because it propagates backwards to younger instructions
  - This is why we can use instruction queues instead of just fetch stage
- Usually if there's 2 or more instructions need to use the CDB in the same cycle, we would prioritize older instructions
- Summary – Tomasulo's Algorithm can overlap loop iterations

Why Can Tomasulo Overlap Iterations?

- Register renaming – multiple iterations use different physical destinations for registers
- Dynamic loop unrolling
- Reservation stations – buffer old values of registers, and avoid WAR hazards
- Tomasulo builds a data flow dependency graph on the fly

Two Major Advantages

- Distribution of hazard detection logic
  - Distributed reservation stations and CDB
  - If multple instructions are waiting on a single result, and each has operands, then they can be issued simultaneously be issued once the result is broadcast on the CDB

Dynamic Scheduling Summary

- Decreasing CPI by increasing throughput (IPC)
  - Higher pipeline/FU utilization
- Split decode into dispatch + issue
- Tomasulo – copy-based register renaming, full out-of-order execution

Precise State and Speculation

- Register renaming now – true renaming + ROB

Speculation and Precise Interrupts

- Sequential semantics for interrupts
  - All instructions before interrupt should complete and all instructions after should look like they never started
  - Same for branches!
  - Precise interrupts return to a precise state
- What makes precise interrupts difficult?
  - Undo post-interrupt (by extension, branches) writeback
  - Not an issue in-order processors as branches complete before younger instructions writeback

Precise State

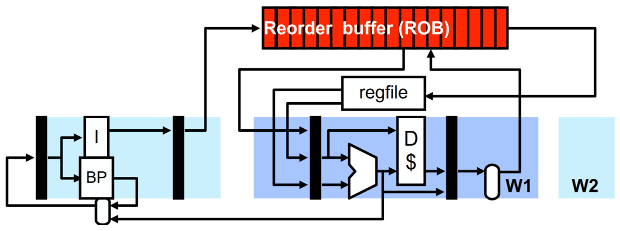
- Speculative execution requires ability to abort and restart at every branch
- Precise synchronous (prog-internal) interrupts require ability to abort and restart at every load, store, div, etc.
  - "Exceptions"
- Precise asynchronous (external) interrupts require ability to abort and restart at every instruction
- Ergo, we should implement ability to abort and restart at every instruction

  - This is called a precise state

- Problem with precise state – writeback combines two separate functions
  - Forwards values to younger instructions – ok if OoO
  - Writing values to the architectural registers
    * Want this to be in-order
- We saw similar problems with decode for OoO execution
  - Solution: split decode into in-order and dispatch and OoO issue

Reorder Buffer (ROB)

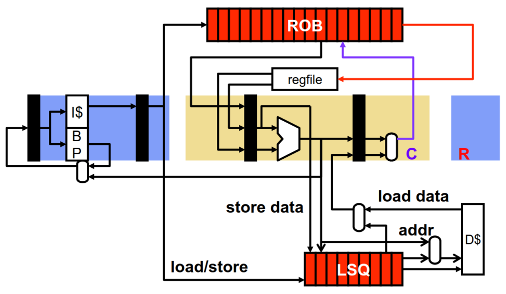
- Conceptually, the ROB is a FIFO queue containing:
  - Flag indication completion
  - New and old register mapping



- Instruction buffer – reorder buffer
- Conceptually, buffers completed results en route to the register file
- Split writeback into two stages
  - Complete (C) – completed item writes result into physical register file, wait doesn't backpropagate
  - Retire (R) – aka commit, graduate – RoB returns to old value to free list, happens in-order
    * Could be a stall if ROB queue filled up

Load/Store Queue (LSQ)

- ROB makes register writes in-order, what about stores?
  - As usual? i.e., do D$ in execute – NO!!!!



- Imprecise memory is worse than imprecise registers
- Allow completed stores to write to the lSQ
- When stores retire, will be at the head of the lSQ, written to D$/memory
- When a load executes, accesses LSQ and D$ in parallel to find correct version of data it should be loading
- Forwards value found in LSQ if there is an older store with a matching address
  - If found, returns value stored by the youngest older store
- Writes only happen when store is at head of both ROB and LSQ

Register Renaming in R10K

- Architectural register file? Gone, names only!
- Physical register file holds all values
- Number of physical registers > number of architectural registers
  - Removes WAR/WAW hazards entirely
- Fundamental change to map table – mapping can never to 0
- Free list keeps track of unallocated physical registers
  - ROB decides who gets freed – ensures precise state
- Conceptually this is true register renaming

R10K Data Structures

- New tags
  - With Tomasulo it was either 0 or RS ID
  - In R10K, is RS#
- Reorder Buffer
  - T – physical register corresponding to instructions' logical output
  - Told – physical register previously mapped to this instructions' logical output
- Reservation station
  - T, 1, S2: output, input physical register #
  - + indicates ready bit
- Map table – T+ holds PR# – never NULL + ready bit
- Free list – physical registers that can be allocated
- No values in ROB, RS, or CDB

R10K Pipeline

- R10K pipeline structure – F D S X C R
- Dispatch
  - Structural hazards possible on RS, ROB, physical registers
  - Allocate new RS and ROB entries, new physical register tags
  - Reads PR tags for input registers and hold in RS
  - Read old PR tag for output reg and store in ROB
  - Store new output tag in ROB and RS
- Execute
  - Free RS entry at execute
  - Can be easier b/c RS# is no longer a tag
- Complete
  - Write destination PR
  - Set instruction output register ready bit in map table
  - Set ready bit for matching input tags in RS
- Retire
  - ROB head not completed? – stall
  - Handle exceptions
  - Some edge cases
    - If instruction at head is a store, write value from LSQ into D$/mem
  - Free previous
  - Free ROB (& LSQ) entry
  - Free previous physical register Told
- Unlike Tomasulo, we'll see that we can issue only 1 instruction in any cyce as it needs to read from the register file

Precise State in R10K

- Physical registers are written out-of-order in C
- This is OK – no architectural register file
- We can "free" written registers and "restore" old ones
- Manipulate the map table and free list

Restoring Precise State in R10K

- Option I – serial rollback using T, Told
  - Is slow, but simple
- Option II – single-cycle restoration from checkpoints
  - Faster but checkpoints are expensive in hardware
- Modern processors compromise – make common case fast
  - Checkpoint only low confidence branches
  - Serial recovery for page faults and interrupts

MIPS R10K Summary

- True register renaming via physical register file
- At retirement, free old register mapping
- Avoid extra data copyin
- Don't move data from register to reservation station
- Rollback
  - Undo speculative instructions
  - Support precise exception model

Superscalar + Out-of-Order + Speculation

- Go great together
  - CPI ≥ 1 – go superscalar (later)
- Superscalar increases RAW hazards
  - Go out-of-order
  - Still a problem? Build a larger window
- Are branches a problem?
  - Add control speculation

6 Caches

Memory Wall

- Processors get faster more quickly than memory
  - Processors – 35% to 55%
  - Memory – 7%

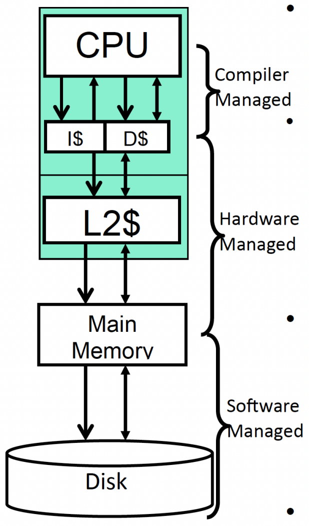
Motivation

- Processor can only compute as fast as memory
- Unattainable goals:
  - Memory that operates at processor speeds
  - Memory as large as needed to run all programs
  - Cost-effective memory

Locality to the Rescue

- Locality of memory references – property of real programs
- Book and library analogy
- Temporal locality – recently referenced data is likely to be referenced again
  - Reactive – cache recently used data in a small and fast memory
- Spatial locality – more likely to reference data near the recently referenced data
  - Proactive – fetching data in large chunks to include nearby data
- Upper components are small and inexpensive
- Lower are bigger and have more latency
- Connected by buses, can have own bandwidth and latency issues
- Most frequently accessed data in M1
  - Next most frequently accesses would be in M2, M3, etc.
- Optimize average access time
  - $\text{latency}_{avg} = \text{latency}_{M1} + \%_{miss} \cdot \text{latency}_{M2}$
- Want to maintain illusion that memory performance is matching CPU performance

Concrete Memory Hierarchy



- 1st level – primary caches
  - Split I$ and D$ caches
  - Typically 8-64 KB, access 1-2 cycles
- 2nd level – second-level cache
  - On-chip, make use of SRAM (same as CPU)

  - Typically 512 KB – 16 MB, access is 5-10 cycles
- 3rd level – main memory
  - Made of DRAM
  - Typically 16 GB to 32 GB, access time is >100 cycles
- 4th level – disk spaces
- Coming up – cache organization

Basic Memory Array Structures

- 2^n entries where n is number of address bits
- Decoder changes n-bit address into 2^n one-hot signals
- 1-bit signal travels on wordlines and activates selected bits

Caches: Finding Data Via Indexing

- To find entry: decode part of address
- 32B blocks – 5 lowest bits
- 1024 lines – 10 next lowest bits

Caches: Knowing That You Found It

- Each entry above can hold one of 2^{17} blocks
- How to know which if any block is at a given index?
  - To each entry, attach a tag and a valid bit
  - Compare tag bits to the address bits
- Lookup algorithm
  1. Read entry indicated by index bits
  2. "Hit" if tag matches and valid is high
  3. Else, a "miss" – fetch block and "fill"

Calculating Tag Overhead

- 32 KB cache only means 32 KB of data
  - Tag storage is overhead!

Handling a Cache Miss

- Cache controller – FSM
  - Remembers miss address and accesses next level of memory
  - Wait for responses and writes
  - Happens on the fill path

Cache Performance Equations

$$\%_{miss} = \#_{misses}/\#_{accesses}$$

- t_{hit} is the time to access the cache on a hit
- t_{miss} is the time to get data into the cache
- Performance metric – $t_{avg} = t_{hit} + \%_{miss} \cdot t_{miss}$

Set-Associativity

- regardless of block size, could still have pairs that continually evict each other
- fix into organizing a set associative cache
- block can reside in one of a few lines
  - line groups called sets
  - each line in a set called a way
- 1-way also called a direct-mapped cache
- 1-set is also called a fully associative cache
- set-associative caches fall somewhere in between
- lookup algorithm
  - use index bits to find the set
  - read data/tags in all lines in parallel
  - if any match and valid bit is set, then hit
- plus we can avoid conflicts, previously conflicting addresses can coexist
- but notice that address rearrangement caused some conflicts
- usually we'll find that avoidance usually dominates
- also we see increased latency due to extra comparator logic

Replacement Policies

- new design choice – on a cache miss, which bloc should we be kicking out (evicting or replacing)
- some options
  - random – doesn't capture locality
  - FIFO
  - least recently used
  - not most recently used
  - Belady's algorithm – replace the block that will be used furthest in the future

Cache Optimization: Prefetching

- reducing miss penalty via parallelism
- can be done via hardware or software
- prefetching – put blocks into the cache speculatively, the key is to anticipate upcoming addresses accurately
- simple example – next block prefetcher
  - works for instructions – sequential execution
  - works for data – arrayed data

Prefetching Metrics

- timeliness – initiate prefetches sufficiently in advance
  - was the data placed into the cache before it was needed?
- coverage – prefetch as many misses as possible, ideally all
  - how many misses did you save?

$$\text{coverage} = \frac{\text{number of useful prefetches}}{\text{number of misses with no prefetching}}$$

- accuracy – don't pollute cache with unnecessary data, might evict useful data

Software Prefetching

- binding – prefetch data into a register
- non-binding – prefetch into the cache

Hardware Prefetches

- what to prefetch?
  - could do > 1 blocks ahead to hide more latency
  - address-speculation – needed for non-sequential data
- when to prefetch? – another design choice
  - ideally want to prefetch when resident block becomes dead
- correlating predictor
- stride prefetcher

Write-Through vs Write-Back

- write-through – immediately on hit, update the cache and next level of cahce/memory
- write-back – only when block is replaced
  - requires an additional "dirty bit" per block
  - replacing a clean block: no extra traffic
  - replacing dirty bloc: no extra traffic

- writeback buffer – keep it off criticial path of miss
  - send fill request of miss to next level of cache
  - while waiting, write dirty blok into the buffer
  - when ew block arrives, put it into the cache
  - writeback buffer writes content to the next level

Comparison of Write Propagation

- write-through
  - conceptually simpler (no WBB)
  - uniform latency on miss
  - requires additional bus bandwidth (think about repeated write hits)
  - sometimes used by L1 caches
- write-back key advantage is that it uses less bandwidth

Write-Allocate vs Write-Non-Allocate

- write-allocate – read block from lower level, write into cache
  - decrease read misses but needs more bandwidth
  - used mostly w/ write-through

Classifying Misses

- compulsory (cold) – when the processor has never seen that address before (would miss even in an infinite cache)
- capacity – cache is too small, would miss even in a fully associative cache
- conflict – associativity is too low, found by taking total misses minus cold + capacity misses
- fourth type (coherence) can only happen in shared memory multiprocs

Miss Rate: ABC

- target cache parameter to fix a specific miss reason
- associativity – decrease number of conflict misses
  - other types not affected
  - increase hit latency
- block size – increase conflict misses bc fewer blocks
  - decrease compulsory misses by leveraging spatial locality
  - no significant impact to hit latency but could require more power
    - could increase miss latency
- capacity – decreases capacity misses but increases hit latency

Capacity and Performance

- simplest way to reduce miss rate – increase capacity
  - miss rate decreases monotonically but with diminishing returns
- t_{hit} increases
  - latency is proportional to $\sqrt{\text{capacity}}$

Block Size

- given capacity – one option to increase block size
- double block size – one less index bit
- reduce miss rate to a point

Effect of Block Size and Miss Rate

- Two main effects
  1. spatial prefetching (good) for blocks w/ adjacent address
     - turns miss/miss into miss/hit blocks
  2. interference (bad) for blocks w/ non-adjacent but in adjacent sets
     - turns hits into misses by disallowing simultaneous residence
- benefit is very program-dependent but good size is 16-128 bytes

Associativity and Performance

- higher associative caches
  - have better miss rate – diminishing returns for single-threaded applications
  - hit time increases, average time depends
- block size and number of sets should be powers of two for easier indexing

7 Multiprocessors

- three main issues
  - cache coherence – how to make cache writes appear at other caches?
  - synchronization – how to regulate accesses to shared data? how to implement critical sections?
  - memory consistency – how do I keep programmers sane while letting hardware optimize program execution?
    - also how to reconcile shared memory with OoO execution?
- two cache coherency approaches?
  - snooping (aka broadcasts) – SMPs < 16 procs
  - directory/scalable – lots of procs

Flynn's Taxonomy

- 2D taxonomy of parallelism
  - num of instruction streams
  - num of data streams
- SISD – single instruction, single data
  - pipelining and ILP on a uniproc
- SIMD – single instruction, multiple data
  - data-level parallelism
  - vector procs/insns, GPUs
- MIMD – multiple instruction, multiple data
  - SPMD – single program multiple data is a part of this
  - thread-level parallelism and on parallel procs
- MISD – multiple instruction, single data
  - NASA????

Thread-Level Parallelism

- collection of async tasks – not started and stopped together
- data shared dynamically and loosely

Shared Memory

- multiple execution contexts sharing single address space
  - MIMD/SPMD
  - implicit communication via LDs/STs
  - complex HW – create a unified view of memory
- provide abstraction which is familiar and efficient for programmers

Hardware Cache Coherence

- rough goal: all caches habe same data at all times else don't have it at all
- minimize cache flushing and maximize caching

Bus-Based Multiprocessors

- all MPs use a bus – all procs see all requests for data at the same time in the same order
- single or banked memory modules
- coherence: all copies kept in the caches have the same data at all times
- cache controller: examines but traffic to find out out who is asking and for what
- executes cache coherence protocol
  - what to do with local copy of a block when you see different things happening on the bus

Coherence for Writeback Caches

- 3 proc-initiated events
  - read
  - write
  - write-back
- 2 bus-initiated events
  - bus-read
  - bus-write
- one response event – SD which is send data
- point-to-point network protocols – typical solution is a directory protocol

Valid-Invalid Coherence Protocol

- two-states per block
  - valid – have block in cache, modified when block is dirty
  - invalid – don't have block in cache
- implemented with a valid bit

MSI – A Realistic Coherence Protocol

- VI only allows one copy in the entire system, multiple copies can't exist, even if there are only readers
- MSI – modified-shared-invalid, valid gets split in 2
  - modified – local copy is dirty
  - shared – local copy is clean
  - if in modified, only the writing cache is allowed to have it, everybody else is invalid
- write from shared is also called an upgrade miss
- modified to shared transition will also write back to memory

Coherence and Misses

- coherence introduces 2 new types of misses
- upgrade miss – delay to acquire write permission to a read-only block
- coherence miss – miss to block evicted by a bus event

Cache Parameters and Coherence Misses

- larger capacity – more coherence misses but offset by fewer capacity misses
- assumed rate of mutable data stays constant
- increased block size – "false sharing" so more coherence misses
  - sharing a cache block w/o sharing data
  - creates a pathological ping-pong behaviour
- more procs – more coherence misses

Scalable Cache Coherence

- part I – bus bandwidth
  - replace non-scalable bandwidth substrate w/ scalable ones
  - point-to-point networks like a mesh
- part II – processor snooping bandwidth
  - interesting – most snoops will result in no action
  - for loosely cached data, other processors probably not calling blocks
  - replace non-scalable broadcast protocol w/ scalable directory protocols

Directory Coherence Protocols

- physical address space statically partitioned
- can easily determine which memory module holds a given block
  - using address bits called the home node
  - can't easily determine which procs have block in their caches
- bus-based protocol – broadcasts events to all procs/caches, simple and fast but not scalable

Directories

- for each physical cache line whose home this is, track
  - owner – which procs have a dirty copy (M)
  - sharer – which procs have a clean copy (S)
- procs send coherence events to home directory

Directory Flip Side: Latency

- lower bandwidth consumption and thus more scalable, but have longer latency
- 2 situations
  - unshared block – data from mem
    - bus: 2 hops (P0->mem->P0)
    - directory: 2 hops (P0->dir->P0)
  - shared block – data must come from another proc
    - bus: 2 hops (P0->P1->P0)
    - directory: 3 hops (P0->dir->P1->P0)
- with memory procs and loosely shared data, then high probability of second case
- there is also the MESI protocol with an exclusive shared state

Synchronization

- answers the questions
    1. how to regulate access to shared data?
    2. how to implement critical section?
- motivation – certain accesses should appear atomic, should not interleave
- not exclusive to multiprocs, but can happen even on a multiprogrammed uniproc
- goal is to regulate access to shared data
- SW constructs – semaphores, monitors
- HW primitives – locks
    - can either acquire or release a lock
    - region b/w is called a critical section
    - must interleave acquire and release
    - 2nd acquire will fail

Working Spin Lock: Exchange

- ISA provides an atomic lock acquisition instruction
- atomically executes the following
    - `mov R1 -> R2`
    - `lw [&lock] -> R1`
    - `sw R2 -> [&lock]`
- if lock is 1, was initially busy, keep looping
- if lock is 0, was initially free, sets the lock

Atomic Exchange Implementation

- need to ensure no interleaving of memory operation
- requires blocking address by other threads
- how to pipeline???
- we also have the load lock (LL) and store conditional (SC) instructions
    - on an LL, the proc remembers the address ad looks for writes by other procs
    - on an SC, any writes are detected to that same address get annulled

Locks and Coherence

- every time a proc tries to acquire a lock it will do a store
- we will see lots of invalidation traffic, limited BW use
- spin locks, spin and read on local copy try to update when lock is available
- solution is test and test and set locks

Memory Consistency

- main questions
  1. how to keep SWE sane while letting HW optimize?
  2. how to reconcile shared memory w/ OoO execution
- memory coherence – creates globally uniform (consistent) view across procs of a single memory location (a cache line)
- not enough, cache lines can be individually consistent but with other procs they can be inconsistent
- memory consistency – creates globally consistent view of memory locations relative to each other

Sequential Consistency

- formal definition of memory view programmers expect
- procs see own LD/ST in program order, this is provided naturally even in OoO execution
- all procs will see each other's LD/ST in program order
- finally, all procs will see same global order LD/ST order across all memory locations
- last 2 conditions are not naturally enforced by coherence

Enforcing SC

- definition – all LD/ST are naturally ordered
- translation – coherence events for all procs are naturally ordered
- when do coherence events happen naturally?
  - ST – on retirement, enter store buffer, so it happens in order and is OK
  - LD – on execute, very BAD in OoO

Is SC Really Necessary?

- most closely matches programmers' expectations
- restricts performance HW optimization
- what if we retain only some of the requirements
  - all procs see each others LD/ST in program order
  - but they don't have to agree on the same global order
- don't confuse programmers too bad, correctly synchronized programs will still work as expected

Relaxed Consistency

- allows reads and writes to complete out of order
- use synchronization as a mechanism to enforce ordering across LD/ST
- we have the 4 rules of R-R, R-W, W-R, W-W
- the type of relaxed model will define which of the 4 can be relaxed
- allows procs to have in-order store buffer and maintains order in b/w writes

Weak Memory Ordering

- for properly synchronized programs only acquire/release must be strictly ordered, because WE define the critical section
  - b/w critical sections – data is pricate, globally unordered access is OK
  - in critical sections, access to shared data is exclusive anyway, globally unordered access is also OK
- weak ordering – Alpha, IA-64, PowerPC
  - ISA provides fences to indicate scheduling barriers
  - use sync library, don't write your own

8 Superscalar

- various types of parallelism
  - thread – multiple programs on multiproc
  - data – single instructions, multiple data
  - pipeline – execute on insn while decoding insn
- back to ILP – execute multiple independent insn fully in parallel, limited multiple issue
- assuming DIC is constant
  - reduce cycle time – less combinational logic per stage, or more stages
  - reduce CPI – are we limited by ideal CPI=1?

Scalar Pipeline and Flynn's Bottleneck

- Flynn's Bottleneck is CPI=IPC=1
- limit can never even be achieved

Multiple Issue Pipeline – Superscalar

- wider stages need more resources otherwise structural hazards caused by reg/FU/etc
- SW must ensure insn are independent

Terminology

- font-end/back-end parts of the pipeline
- N-way – # of instructions brought into the pipeline
- can be static (IO) or dynamic (OoO)

How Much ILP is There?

- compiler tries to schedule code to avoid stalls
- even given unbounded ILP, superscalar has limits
  - CPI versus clock frequency
  - N=3 or 4 is reasonable N for superscalar
  - Apple's M1 is speculated to be 8-way

Superscalar CPI Calculation

- base CPI for N-way is CPI = 1/N
  - amplifies stall penalties
  - assumes no data stalls

Front-End Challenges

- wise instruction fetch
  - modest – multiple instructions per cycle
  - aggressive – predict multiple instructions
- wide instruction decode – replicate decoders
- lots of challenges with branching and prediction
- wide instruction issue – determine when can proceed in parallel, not all combinations are possible
- wide register read – multiple read ports

Back-End Challenges

- wide execution – replicate FUs, multiple cache ports
- wide writeback – one write port per issue
- wide bypass paths – more possible bypass paths $O(N^2)$ for N-wide machine

Wide Fetch – Sequential Instructions

- what is involved when fetching multiple instructions by cycle
  - in the same block? easy, large block size
    - compiler aligns blocks to cache lines (pad with NOOPs)
  - in different blocks? fetch block A and A+1 in the same cycle

Wide Fetch – Non-Sequential Instructions

- begs 2 questions:
  - how many branches predicted per cycle?
  - can we fetch from multiple taken branches per cycle?
- simplest answer is 1 and No.
- compiler can help here too – unroll loops and redice branch taken frequency

Wide Decode

- each is fixed length, doable if variable length
- need 2N read ports and 1 write port
  - recall latency, power, area scale by N^2
- in reality we can get away with fewer ports
  - read – not all insns need source operands and many can come from bypass networks
  - write – not all write to a source register

Wide Execute

- do we always need to scale by N?
  - N ALUs is okay, as they are small
  - probably not a good idea to have N FP dividers
- typically some mix of FUs that is proportional to DIC
- could see structural hazards but hopefully rare

Wide Memory Access

- we will need N^2 bypass and stall logic
- actually this is ok – these are 5-bit and 1-bit quantities

Clustering

- mitigates N^2 bypass
- group FUs into K clusters
- limit bypass between clusters
- $\frac{N}{k} + 1$ inputs at each mux
- $(N/k)^2$