Performance

- Can det from various metrics, but isn't easy
  - scientific sims. - fp performance
  - prog. dev. - int performance
  - commercial work - mem + i/o
- Response Time - minimize elasped time for program (time_end - time_start)
  - important to computer system user
- Throughput - maximize completion rate = jobs per second.
  - important to computer manager.
- Without overlap in processing throughput = 1/av. resp time, usually throughput > 1/av resp time otherwise.
- Performance for comp arch. is program run time (which we minimize)
  - performance = 1/time, time = resp time, CPU run time, etc.
  - elapsed time = CPU exec time + I/O wait, comp arch focuses on exec time.
  - ↑ performance
    - x faster cpus = ↓ resp time ↑ throughput
    - x more CPUs = ↑ throughput ↓ resp time is possible.
  - Machine is n times faster than B if perf(A)/perf(B) = time(B)/time(A) = n and x% faster if = 1 + x/100.
  - broken into instructions + cycles.
- **Iron Law** — time/program = instr/prog · cycles/instr · sec/cycle
  - sec/cycle (cycle time/clock time) - det by tech + CPU orgn.
  - cycle/instr (CPI) - det by ISA + CPU org., effective CPI not actual, overlap ↓ CPI, average over instr.
  - instr/program (instruction count) - instr executed (not static), det. by program, compiler, + ISA
- Minimize overall time not just individual terms.
- Other metrics include MIPS + MFLOPS, but aren't as accurate.
  - (MIPS - mil instr/sec), (MFLOPS - mil float ops per sec)
  - can be optimized but still worsen performance (is only CPI metric)
  - used in peak perf, MIPS is ok when instr/prog is constant w/ same compiler + ISA.
- Benchmark programs are used to measure + predict workload performance.
  - best case is running same set of progs everyday, but not realistic.
  - SPEC 2017 most recent - (System Performance Evaluation Cooperative) has 12 int + 17 float progs.
    - normalize runtime w/ a Gold standard processor (SPEC ratio)
    - geo mean of the normalized times

Performance Cont'd

- How to average runtimes across programs when benchmarking:
  - Arithmetic mean - Σ time/n (n programs)
    - x valid if programs run equally, otherwise use weight
      Σ weight · time / n
  - Harmonic mean - $\frac{1}{\Sigma \text{rate}/n}$ - for using rates & ending with rates.
  - Geometric mean - $\sqrt[n]{\Pi \text{ratio}}$ - for using ratios
    - x not prop. to total time & usually mis predicts for ≥ 3 machines.
  - Unnormalized raw runtimes are always best.
- Amdahl's Law - make the common case fast
  - Speedup = $\frac{\text{time}_{old}}{\text{time}_{new}} = \frac{\text{unaff}_{old} + \text{aff}_{old}}{\text{unaff}_{new} + \text{aff}_{new}} = \frac{1}{(1-f) + \frac{f}{s}}$ (s factor of speedup, f affected frac)
  - $\lim_{s \to \infty} \frac{1}{1-f+\frac{f}{s}} = \frac{1}{1-f}$ (frac matters for max speedup)

ECE 437 | Chp# 2

Instructions

- words of a computer
- Instruction Set Architecture (ISA) is the vocabulary & defines the interface of computers.
- General purpose computer
  - Fetch (PC) → Execute → Update PC (back to Fetch)
  - PC stores address of instruction & is updated to point to the next instr. to execute.
- Terms
  - Opcode - specifies type of op
  - operands - in & out data (must be registers)
  - multiple instrs per C statement often
- Registers & ALU ops
  - RISC-V uses 32 registers x0 - x31 (32 bit/word)
  - registers are useful for small storage for ops & fitting specifiers in instructions.
  - memory load/store used to access & modify more data.
    - address is byte address so word addressing (Groupings) ends addr. in two 0s in binary. (4 bytes per loc.)
    - using load & store w/ offset is easy way for arrays
      - lw x5 0(base addr) loads A[0]
      - sw x5 4(base addr) stores to A[1]
  - ALU ops ex: add, sub, mul, div, or, and, sll, etc
    - can be done w/ immediates (constants) too
    - ↑ shift left
- Endian determines MSB @ 0x...00 (Little endian) or 0x....11 (Big end.). x86 is little endian.
- Branches & Jumps - used to move around exec path
  - branches compare two registers and either jump to an addr or continue regular execution.
  - jumps move the addr to some other PC and sometimes stores the PC for the next instr. in rd (jal or jalr).

Instructions cont'd

- Procedure Calls
  - Initial responsibilities
    - save regs, setup params., call procedure, get result, restore regs.
  - Process responsibilities
    - do func, set up results, restore more regs, return (save more regs)
  - jal is only hardware instr, rest is software.
- Stack - useful for calls
  - grows from large to small addr, w/ x2 being stack pointer
  - push x5 (addi x2, x2, -4, sw x5, 4(x2)) (cost in basics)
  - pop x5 (lw x5, 4(x2), addi x2, x2, 4)
- Software layers - possible bc prog. can be treated like data.
  - executable: data in → data out
  - loader: executable file → executable in mem.
  - linker: obj file → exe file
  - assembler: asm file → obj file
  - compiler: HLL file → asm file
  - editor: commands → HLL file.
- RISC-V machine language uses 32b instrs., while assembly instrs are wordier.
- Instruction format - regs are 5b, opcode is 7b. (rs is source reg, rd is dest. reg.)
  - R-format - funct7, rs2, rs1, funct3, rd, opcode
    - used for three register ops. mostly simple operations
  - I-format - immediate [11:0], rs, funct3, rd, opcode
    - used for immediate (constant) ops and loads. imm. is signed 2's complement. imm. used for operand or load offset.
  - S-format - imm [11:5], rs2, rs1, funct3, imm [4:0], opcode
    - used for stores
  - B-format - imm [12|10:5], rs2, rs1, funct3, imm [4:1|11], opcode
    - used for branches, 0th bit not used bc 32b instrs. + 16b instrs exist so only one bit not specified.
  - U-format - imm [31:12], rd, opcode
    - used for lui, load upper imm.
  - UJ-format - imm [20|10:1|11|19:12], rd, opcode
    - used for jumps
  - instrs structured weird so bits line up in every instr.. Bits allow us to decode control signals to exec instr.

Instructions cont'd

- Addressing Modes
  - register addressing
  - base addressing - base addr + offset
  - Immediat addressing - immediate values
  - PC relative addressing - address based on PC + immediate
  - Pseudo direct addressing - address in instruction.
  - Indexed addressing - add two ress base + index
  - Indirect addressing - M[M[addr]]
  - Auto incr/decr - add data size
  - Auto update - like displacement/offset but update reg.

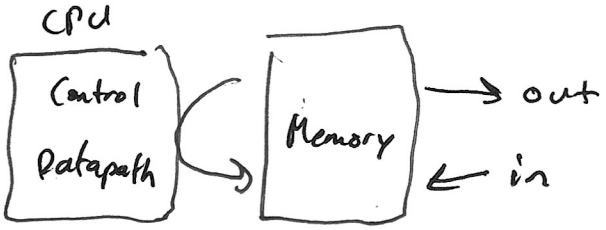
RISC-V

Other ISAs

- Choose ISA based on hardware implementation + optimized compilation.
  - ↓ mem uses dense instr ISAs usually, w/ complex + irregular instrs.
  - ↑ speed uses pipelining (v important) so simple + regular is important.
- Intel 80386 (x86) ISA is a decoding nightmare w/ 1-17 byte instrs. Prefix + postfixes, + weird format, but good core instrs.
  - all instrs have to work in x86 tho.
  - Intel chips translate into RISCy ops for execution units due to backwards compatibility + complex decoding
    X helps to speed up execution + parallelism.
- More powerful/complex instr. doesn't mean faster.

Single Cycle

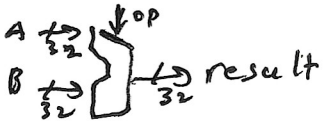
- CPU/processor -

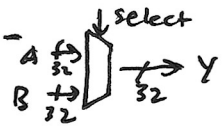


- Datapath for instructions (single cycle) - compose w/ mux, ff, gates, ALU, & reg file.

  - Comb. logic elements -

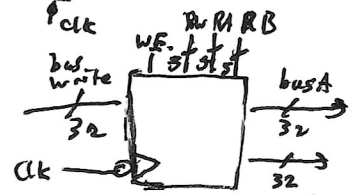
    * adder - 

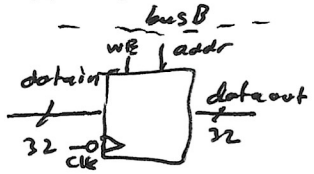
    * ALU - 

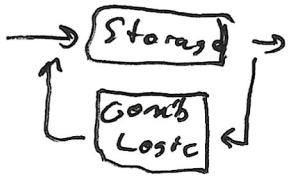
    * MUX - 

  - Storage Elements

    x register (PC) 

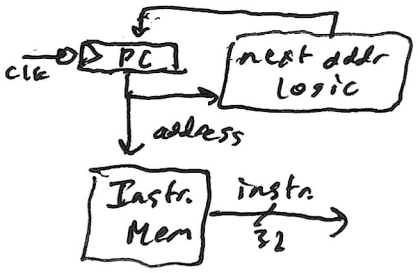
    x register file
      - 32 regs, 2 read ports, 1 write port. 

    x Memory
      - 1 in bus, 1 out bus, not bidirectional 

- Computer: 

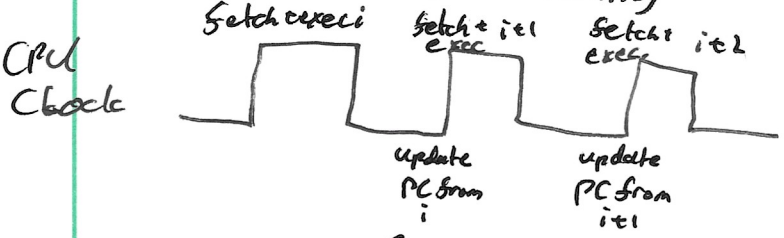
- Single Cycle datapath needs to do entire instr in 1 long clk cycle.
  - fetch, decode (read operands), execute, memory, writeback
  - RTL is useful to represent steps & dp. elements.

Single Cycle cont'd

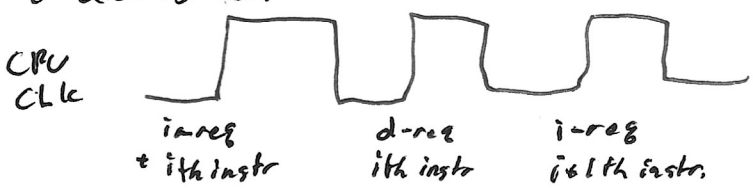
- Register Transfer Language (RTL) is nice short hand similar to pseudocode ~~for~~ which helps describe what instructions do.
  - ex. R[rd] ← R[rs1] + R[rs2] PC ← PC + 4 (add)
- Fetching instructions - fetch + update PC at end of cycle
  - 

  - next addr logic w/o jumps or branches:

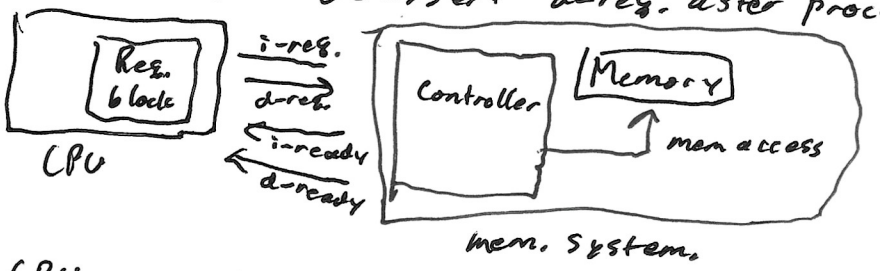

- ALU instructions - utilizes the reg file + ALU (RegWrite + ALUOp)
- Logical ops. w/ immediates - utilizes reg file + ALU + a immediate unit + mux (RegWrite + ALUOp + ALUSrc)
  - mux to switch alu input to immediate
  - Immediate unit to extract + sign extend value.
- Load instructions - uses reg file + ALU to calculate base addr + offset, data memory, + a mux to choose between memory + alu output for writing data to reg file (RegWrite + ALUOp + ALUSrc + MemtoReg)
- Store instructions - uses reg file + ALU to calc. base addr + offset, data mem, and a direct data in line from rs2. (ALUOp + ALUSrc + MemWrite)
- Ideal Single cycle timing



  - some fpgas require multiple clocks for mem. access for arbitration between Instr. fetch + data fetch



Single Cycle Control

- Memory Arbitration
  - Decides on what mem req. to process, as i-reg is generated in all instrs. including d-regs.
  - Prioritizes d-regs > i-regs (i-reg will be fetching next instr. while d-reg. is for current instr.)
    - ✗ will have to deassert d-reg. after processed
  - 

  - CPU mem interface is seq. while controller is combinational.
  - Need to latch instr. while waiting for mem. to prevent instr. vanishing.
  - Needs to work at varying mem. latency (mem access time unknown in # of cycles). Allows to check for timing independence.
- For jump & branch instrs. additional muxes & adders & connections are made to update PC and sometimes registers.
- Control signals control datapath units for each instr based on opcodes, functs, and other parameters.
- Cycle time or max CPU clk speed depends on the design.
  - Time to compute next state values
    - ✗ propagation delay clk → Q (FF)
    - ✗ comb. logic delay
    - ✗ setup requirements for FF.
  - Often times, load instr. have longest critical paths in single cycle due to data mem access.
    - ✗ Leads to very long cycle time & slow processing.

Pipelining

- 1st of 4 big ideas of modern computer systems.
- Pipelining benefits (improving performance) - ↑ throughput, = or ↓ latency
  - Breaking instructions into stages & overlapping speeds up cycle time while keeping CPI ≅ 1.
- Common Pipeline Stages
  - Ifetch - instruction fetch (from mem)
  - regldec - Reg. read & instr. decode
  - Exec - compute/calc. mem. addr
  - Mem - read data from memory
  - Wr - write data to reg. file.
- Pipeline speedup
  - Ideal speedup = # of pipeline stages (equally balanced)
  - Max speedup ≤ # of stages, Speedup ≤ $\frac{\text{Time for unpipelined op}}{\text{Time for longest stage}}$
  - Async / self-timed pipelines don't follow this but aren't common due to difficulty for high perf complex CPUs.
- Pipelining is implemented using latches (FFs) which keep the instrs. and transfer them between stages.
  - ↓ prog. latency by ↑ instr throughput but keeping or increasing instr. latency (due to things like latch overhead)
  - Associate resources w/ states
  - Resources not necc indivisible/full-cycle (write to reg first half, read second half)
  - Carry necessary control signals along.
- Observations
  - Each hardware block only used once per instr. & must be used at same stage every time.
    - X R-types instr. don't use mem. so just do nothing during mem. stage.
- Control Signals - generate once & pass along unconsumed signals through each stage

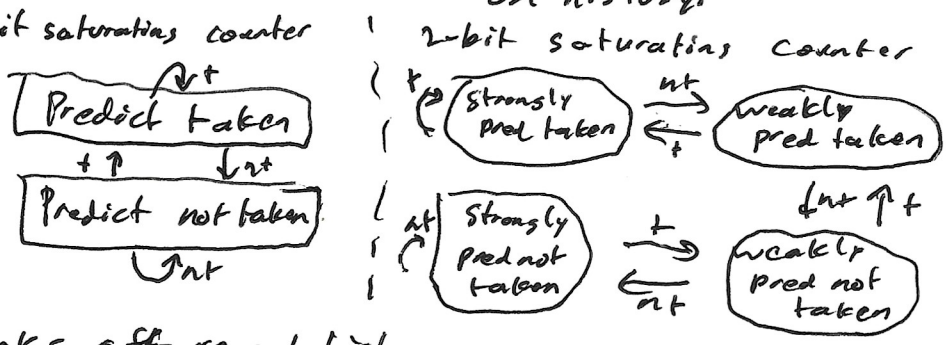
ECE 437 | Chpt. 4b cont'd

Pipelining cont'd

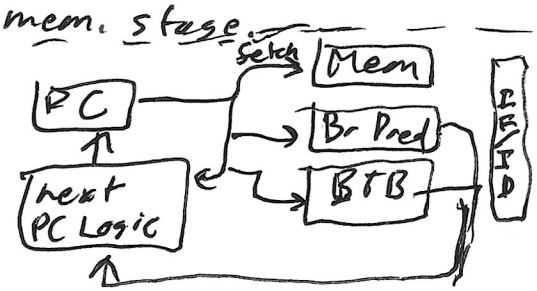
- Pipeline hazards - issues arising from the pipeline system.
  - Must be detected to address
  - Always can be solved by waiting, w/ stalls, bubbles.
    - ↑ CPI, ↓ speedup
- Structural Hazard - 2 instructions need same hardware at same time
  - single memory, solve similar to S.C., process d-req. first.
  - write half cycle reg, read next half cycle.
- Data Hazards - data not ready
  - dependencies backwards in time., true dependence is forward in time
  - very common, so stalling slows down majorly
  - utilize a forwarding unit to forward data from future latches back to exe stage ↑ muxes
  - RAW (read-after-write) (forwarding), WAW (allwb in final stage), WAR (fetch operands early in pipe)
    - RAW only real hazard.
    - RAW detected from pending writes + operands having same regs.
    - WB stage no hazard bc write in 1st half cycle then read.
  - Loads still introduce RAW bc value not avail. till mem stage, need to stall. earlier stages
    - True backward dependency
- Terms
  - Hazard - true backward dependence in time (can't be solved by forwarding)
  - Dependencies - in code, cannot be solved away (not always a hazard)
- Control Flow Hazards - not known which instr. to fetch.
  - branch instrs. + jump instr. (jump is calcd. but may be avoided)
  - somewhat common so bubbles usually too much overhead. lead to major delays.
  - can solve with delay slots w/ compiler putting indep. instrs. in delay slots after branch
    - low use as long pipelines don't effectively use + indep instrs. to be placed aren't very common.
  - jumps will almost always trigger this apart from advanced solutions.

Pipelining cont'd

- Branch Prediction - combats c.f. hazards by choosing a branch to follow.
  - Easiest - predict not taken always (PC+4 easy to calc)
    - x continues execution seamlessly.
    - Need to flush and fetch correct instr. if incorrect (same delay as bubble so no issue in guessing)
  - Dynamic branch prediction - adjusts prediction based on history.



  - x works off recent history
  - x store state of predictor in branch predict table w/ PC used for index + stored as tag (replacement occurs if collision).
  - x store prior targets of branches of taken in branch target buffer in a similar manner to allow access to branch taken PC immediately
  - x update BTB + BPT when instr. is resolved in mem. stage.
  - x



- When calcing CPI do CPI = ideal + frac. stall + ...
  - remember to calc branch % + branch pred error.

ECE 437 | Chpt 4b cont'd

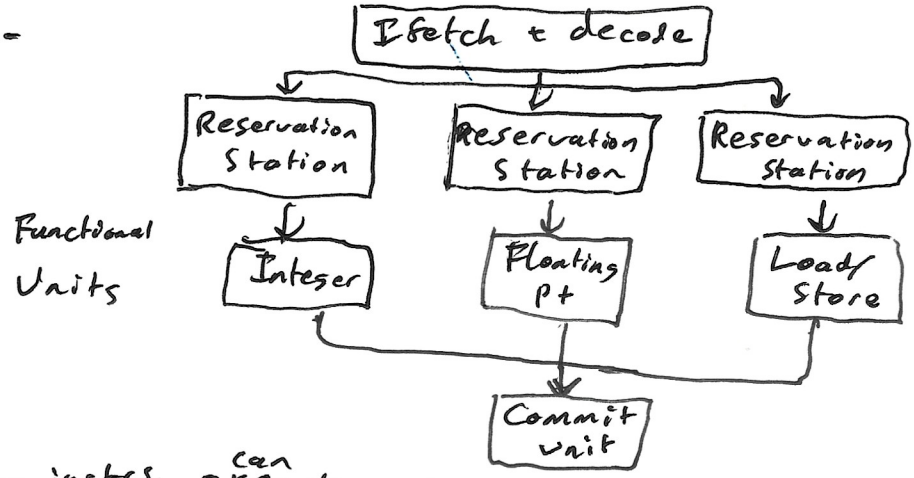
Pipelining cont'd

- Exception - unprogrammed suprise function call (external CRITICAL)
  - handled by OS, must record the address of the offending instruction
  - returns control to user by saving + restoring user state
  - lets OS get in + out without breaking program.
- Interrupts - caused by external events (e.g. I/O)
  - async to pros. exec.
  - Can be handled between instrs.
  - suspend user prog., handle, and resume user pros.
- Traps - caused by internal events
  - caused by exceptional conditions, errors, or faults.
  - sync. to prog execution
  - cond. must be remedied by handler of prog.
  - instr. may be retried/simulated + prog. continued, or pros. may be aborted.

- OS wouldn't work w/o exceptions + interrupts
  - uncommon case but must implement
- RISC-V defines instructions having no effect if it causes an exception.
- Pipeline Considerations for exceptions
  - no instr. after the excepting instr. may execute
  - all instr. before excepting instr. must execute
  - excep. handler software saves + restores regs. and other state.
- RISC-V Exceptions - all exceptions jump to same handler code w/ a cause register.
  - we consider illegal instrs., arith. overflow, + hardware malfunction.
  - Hardware behavior
    - x Save PC of offending instr.
    - x use special reg. SEPC
    - x set SCAUSE reg appropriately: ECC=0, OVF=1
    - x jump to handler at fixed addr.

Pipelining cont'd

- Datapath modifications for exceptions
  - Hardware malfunction - detected in EX, squash (convert to NOP) + in earlier stages
  - Illegal instruction - detected in ID, squash (convert to NOP) + in earlier stages
  - Need exception sorting when overflow followed by illegal exception (causes simultaneous exceptions)
- Superscalar Processor — multiple scalar ops/instrs in one cycle.
  - duplicate hardware for alu, immsea, more lines out of register etc. for multi scalar ops per cycle.
  - Static multi-issue processor
- Dynamic Scheduling
  - helps avoid/reduce effect of hazard
  - execute indep instrs. while waiting on results of prior instrs.
  -



  - instrs. can execute when operands are ready
  - instrs. commit in order when all prior instrs. have committed

Caches

- 2nd of 4 big ideas of modern computer systems
- CPU is majorly slowed by mem. system
  - real mem can take 300+ cycles.
  - large mem is slow + cheap, small mem. is fast + expensive.
  - solved by mem hierarchy of progressively smaller + faster mems.
    - x provides capacity of large mem. + speed of small mem.
- Locality - programs access small portions of address space before moving on.
  - mem. access, data structures, instructions, are all often benefited by taking advantage of their locality.
  - Temporal Locality - Locality in time, keep recently accessed data close to processor.
  - Spatial Locality - Locality in space, move blocks of data to upper levels instead of single entries.
  - Move data to cache on access for temp. locality and move a block instead of a single word for spacial locality into fast mem./cache
    - x As locality changes move new data in + replace old data in cache.
- Terms
  - Hit - data is in upper level (often cache)
    - x Hit Rate - fraction of mem accesses to upper level
    - x Hit Time - Time to access upper level: cache access time + Time to det hit/miss
  - Miss - data needs to be retrieved from block in lower level.
    - x Miss Rate - 1 - Hit Rate
    - x Miss Penalty - Time to fetch block from lower level + Time to deliver block to CPU
  - Hit Time << Miss Penalty.
- Caches take advantage of Amdahl's law, as cache-hits through locality are the common case.
  - Reduces average memory access time (AMAT)
    - x AMAT = hit time + miss rate * miss penalty

Caches cont'd

- Managing the memory hierarchy
  - Register ↔ Main memory - managed by compiler (sometimes programmer)
    - x Word granularity, lw/sw ties mem loc. to registers (alloc), temp. regs to spill mem. when needed.
  - Disk ↔ Memory - managed by OS for mem.
    - x Automatic + transparent to user
    - x Managed by programmer for file r/w
    - x Virtual memory + illusion of large mem. w/ protection.
  - Cache ↔ Memory - managed by hardware. for speed
    - x auto managed
    - x Block granularity for spatial locality, retaining recently used blocks for temporal locality.
- Cache - Hardware Hash Table
  - Tag - indicates which address is in the slot
  - Data - block of data
  - Valid - whether the data in the entry is valid.
  - Block/Cache Frame - each slot in the cache.

  ] Cache Frame
- Hit/Miss detection
  - tag matching → hit + returning data to processor
  - else, miss, evict a block, set block from mem into cache.
- Address: Tag, index, block offset, byte offset
  - Byte Offset - bits to index bytes in each line in the cache block
    - x ex. 4 byte line = 2 b byte offset [1:0]
  - Block offset - bits to index lines in each block
    - x ex. 2 lines per block = 1 b block offset [2]
  - Index - bits to index blocks in cache
    - x ex. 64 blocks in cache = 6 b index [8:3]
  - Tag - leftover bits to store in cache to rep. address.
  - Block offset can include byte offset, ~~but just~~ Since addr. are byte addr.
  - More detailed expl. later.

Caches cont'd

- Block Placement Schemes
  - Direct mapped - many → one - block can only be in one frame
  - Fully associative - many → any - block can be in any frame
  - N-way associative - many → N - block can be in one of N-frame
    - N tags in set searched in parallel.
    - Index selects set, tags are compared, & data is selected.
- Terms.
  - Cache Block - aka cacheline
  - way - one of the assoc. ways in a set
  - set - set of all the ways/blocks in a set
  - index - selects the set
  - frame - physical loc. for 1 cache block.
- Cache Organization
  - Frames = Cache size / block size
  - # sets = Frames / ways
  - Block-offset bits = lg (Block size)
  - Index bits = lg (# sets)
  - Tag bits = addr. bits - index bits - block-offset bits
- Set-assoc. cache adds another mux & have to det hit & miss early.
- Associativity Spectrum
  - Conflict misses ↓ with higher assoc due to cache mapping
  - ↑ assoc. = ↑ tag bits, ↓ idx bits, ↑ comparator size
    - assoc. search is complex.
  - Direct Mapped ⟶ (miss rate ↓) ⟵ (Complexity ↓) Fully Assoc.
  - A little assoc. goes a long way in ↓ miss rate.

Caches cont'd

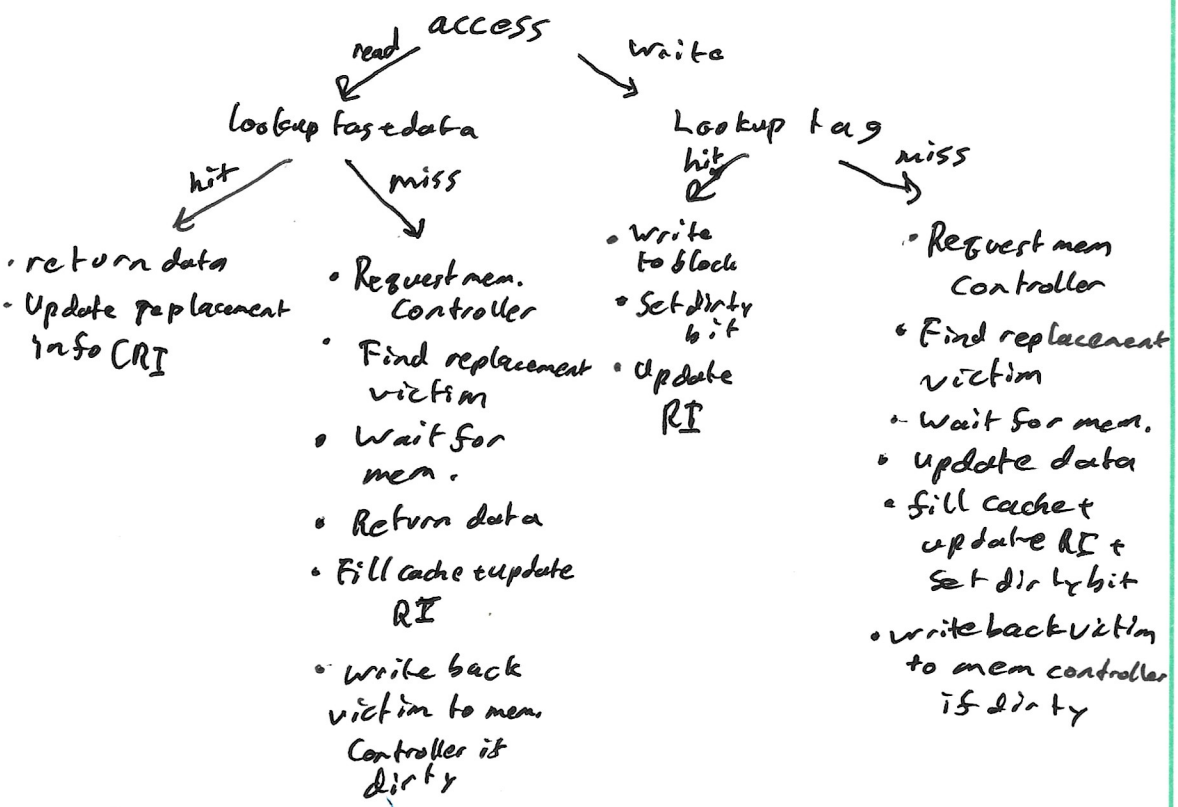
- Miss Types
  - Compulsory - cold start / process migration - first access to every block
    - x can't do much, insignificant if running tons of instrs.
  - Conflict - collision - multiple mem. loc. mapped to same cache loc.
    - x Can ↑ cache size/associativity
  - Capacity - cache can't contain all blocks accessed by the program
    - x can ↑ cache size
  - Coherence - invalidations - other process updates mem (multicore/I/O)
  -

	Direct Mapped	N-way set assoc.	Fully assoc.
Cache Size	Big	Medium	Small
Compulsory Miss	Same	Same	Same
Conflict Miss	High	Medium	0
Capacity Miss	Low	Medium	High
Coherence Miss	Same	Same	Same

- Block Replacement Policies
  - Direct Mapped is easy (have to replace block filling up spot)
  - Assoc. Caches
    - x Random - simple
    - x FIFO - not usually good
    - x LRU (least recently used) - smart
    - x NRU (not recently used) - not bad.
- Handling Writes
  - Could queue up another store while handling the current one, as stores don't write to register.
  - Write-through on writes - write to cache + memory (write buffer)
    - x simple but ↑ mem traffic.
    - x write buffer can be fifo from the store queue
    - x using L2 cache ↓ delays to overflow
  - Write-back ~~on~~ write to cache and write to mem. when block is replaced.
    - x only one write to mem. (write hits)
    - x reads can cause writes (evicting blocks)
    - x multicore coherence (non-intuitive)

Caches cont'd

- Cache operation (WB)
  -



  - seperate mem controller (controls RAM) + cache controller (controls cache operations + states)
  - mem. controller only acts upon C.C. requests.
  - need to latch regs from c.c. to m.c. to prevent large critical path from addr → cache → mem. → cache → pipeline.
    - ✗ slows clock even further than single cycle often.
    - ✗ latch all req + data between controllers
      - data from mem + req. from cache primarily
- Block Size Tradeoffs
  - ↑ block means ↑ spacial locality usually
    - ✗ ↑ block size → ↑ miss penalty as we have to transfer more data
    - ✗ Too little cache blocks in cache also bumps up miss rate (Big block Small cache)
  - ~~asdfasdfasdf~~

	hit time	Miss penalty	Miss rate
Size L → H	inc.	same	dec
Assoc L → H	inc.	same	dec
Blk size L → H	same	inc.	dec
Replacement policy FIFO → LRU	same	same	dec

Caches cont'd

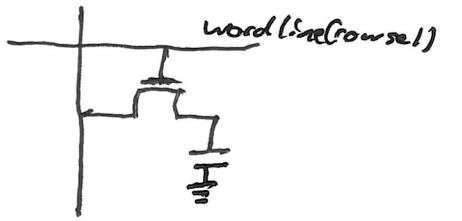
- Improving Cache Performance
  - ↓ hit time → small & simple, ↓ associativity
  - ↓ miss rate → ↑ cache size, ↑ blk size, ↑ associativity
  - ↓ miss penalty → ↓ blk size
- Need to optimize all cache factors and comprise with workloads in mind. (Simplicity often wins)

Cache Factors to optimize.
- Cache size
- Block size
- associativity
- replacement policy
- write back vs write through
- write allocate (bring block in cache on write miss)

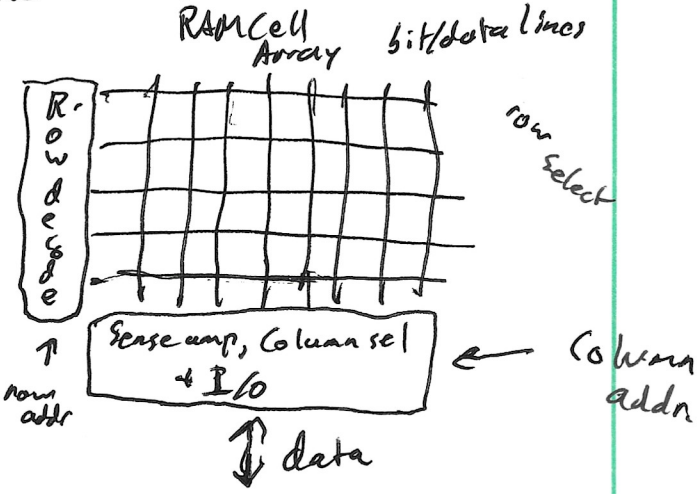
- Multi-level caches ↓ miss penalty since miss penalty of L1 is just the AMAT of L2 instead of RAM latency.
- Split caches for instr & data can avoid structural hazards
  - ~~fastrs~~ instrs are 75% of mem. access.
- Modern CPU caches
  - 3 levels of SRAM cache
  - Low assoc. split L1: i & d cache
  - higher assoc. in L2 & L3, 80% on-chip transistors for caching.
- Memory Storage Elements
  - registers, latches - small, in processor, expensive to add — Processor dp.
  - SRAM (caches) - medium, on chip close to processor, costly
  - DRAM (main mem.) - Large, off-chip, 50ns access time (2006 35-45$ for 512MB) — Mem Subsystem
  - Disk etc. - Large, far from processor, slow (~ms), (2006 70$ for 200GB) — I/O Subsystem
- Random Access - equal time is the same for all loc.
  - DRAM - Dynamic Random Access Mem.
    - ✗ ↑ density, ↓ power, cheap & slow
    - ✗ Dyn: needs to be refreshed regularly even if powered
  - SRAM - Static Random Access Mem.
    - ✗ ↓ density, ↑ power, ↑ cost, fast
    - ✗ Static: content lasts forever if powered.

Caches cont'd

- Not s. RAM (Disk/CDROM)
  - Access time varies from loc. to loc. ± time to time
- Sequential Access Tech - access time linear in loc. (tape)
- DRAM + SRAM (caches + main mem + regs) are not implemented as a bunch of muxed bits
  - Implemented as ~~a bunch of~~ arrays that are read/written
  - DRAM is slow bc of ↑ decode and ↑ array, as well Long off-chip wires (mem. bus) between mem. + CPU.
- DRAM - Dynamic ~~RAM~~ main mem / RAM
  - dense, 1 transistor/cell
  - diff tech from CPU
  - forgets after a while ~(50ms)
  - impl. through ~~wide~~ DIMMs (SIMMs (64b/32b transfers)
  - 1 T DRAM Cell = charge on cap.



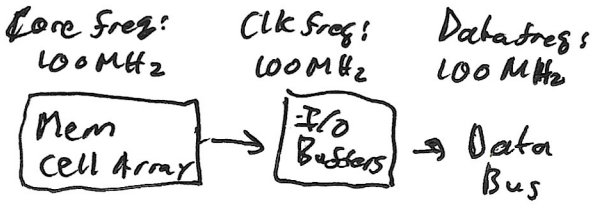
    - ✗ Read:
      - Precharge bitline to VDD
      - Select Row
      - Cell + bitline shares charge
        - cell cap charges at, tiny V change on bline
      - Sense changes (analog) + convert to digital.
      - write - restore val (reads are destructive.
    - ✗ Write
      - Drive bitline
      - Select row.
    - ✗ Refresh
      - Do dummy read + restore while marching down rows.
  - Classic DRAM organization.



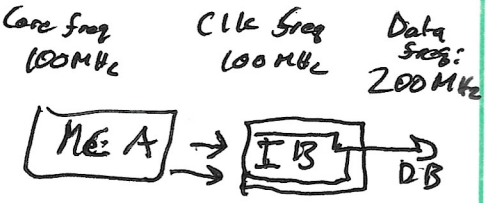
    - ✗ each intersection is a 1t cell
    - ✗ ↑ array = ↑ wires so drain caps on each horz. wire + gate caps on each vert. wire
  - Adding pins is expensive
    - ✗ row + col addr time muxed (same pins, latch vals.)
    - ✗ use 8 chips, 1 bit per chip to retrieve 1 byte.
      (can be expanded to larger values) like 64b. access

Caches cont'd

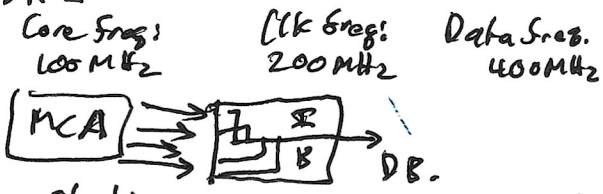
- DRAM Optimizations
  - Fast Page Mode (FPM) - consec. accesses to same row (row hits)
    - row once & vary col. addr. as row access is slow compared to columns (burst mode)
  - EDO DRAM - extending data out - FPM + pipelining of col. accesses
  - Synchronous DRAM - tied to sys clock, ↑ bus speed.
    - SDRAM - DDR, DDR-2, DDR-3
  - RDRAM - rambus - not standard.
- DRAM organization - org/data transfer optimization.
  - SDRAM



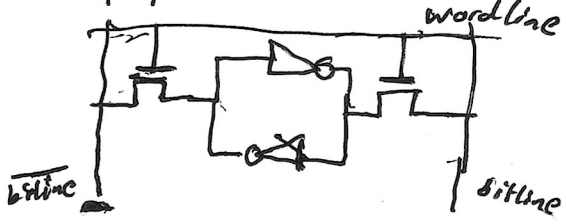
  - DDR1

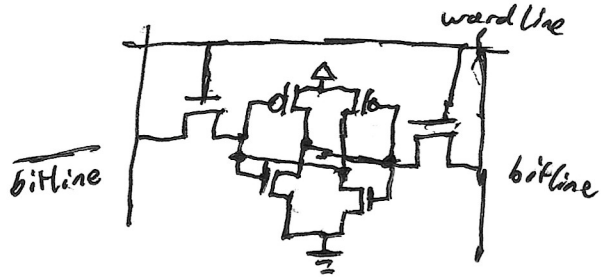


  - DDR2



- SRAM - static, no need for refresh.
  - 6T implementation, on-chip, ~~static~~ CMOS
  - two inverters in a loop to remember w/ access transistors.
  - Slow bc of huge decode + arrays (smaller than DRAM) + long on-chip wires between caches + pipeline.
  - 6T SRAM Cell
    - Write
      - Drive bitlines (bit=1, $\overline{bit}$=0)
      - Select row
    - Read (not destr.)
      - Charge bitlines to VDD
      - Select Row
      - Cell pulls bit or $\overline{bit}$ low
      - Sense amp convert to digital



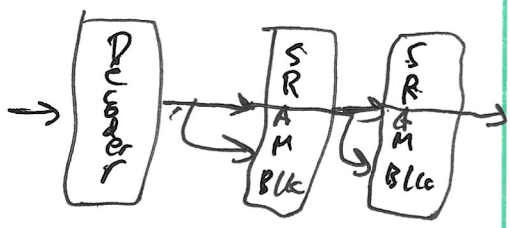


  - SRAM Organization

  - Use similar cap policy to DRAM.

Caches cont'd

- SRAM Organization
  - Arrays might be slightly different
  - fully custom. b.c. no pins + on chip
  - wider access is better since first byte access is slow but consecutive bytes only small incremental cost. (wide access of whole block)



- Bank organization. – allows mass parallel access ↑ bw
  - one huge array is ineff so SRAM+DRAM uses seperate "banks" (arch. level)
  - banks are in chips + concat chips to get access width for DRAM
  - banks divided into sub-arrays for latency/area/power on the circuit level
  - SRAM uses L3 Cache blocks in parallel for bandwidth
  - separate on-chip wires per bank, + access per bank, but still each access is slow (latency)
- Terms
  - Latency – total delay seen
    - Access time – time between req. + word arrival.
    - Cycle time – time between reqs.
  - Bandwidth – size of mem. transfer (L1 + Large Block Miss Penalty (L2))
  - Main mem = DRAM
    - address divided into 2 halves
      - Row access strobe (RAS) or column access strobe (CAS)
  - Cache = SRAM – no addr division
- Cache aware programming
  - Instr. Sequencing
    - Loop interchange: adjust nesting of loops for data access to match memory.
    - Loop fusion: combine 2 indep loops w/ same loop + some overlapping vars.
    - Tiling (Blocking): Improve temp. locality by accessing same blocks instead of accessing rows/cols.
  - Data ~~Layout~~ Layout
    - Merging Arrays: ↑ sp. locality by joining arrays into compound elements.
    - Nonlinear Array Layout: Map 2D array to lin. addr. space
    - Pointer-based Data Structures: node allocation

ECE 437 | Chpt 6

Multicore

- 3rd of 4 big ideas of modern computer systems.
- Why Multicore
  - other improvements ↑ clk speed which means ↑ power
    - $P = CV^2f$ (V linear to f so P ∝ f^3)
  - lower voltage = leakage + reliability at some point.
  - 2x cores = 2x power vs 2x freq = 8x power
- Parallel programming needed for multicore (not hidden from programmer)
  - way harder than seq. programming bc programmers have to track time, or which computation/who produces/needs data when.
- Various Multicore methodologies
  - Shared I/O (Network) - use message passing (distributed systems)
  - Shared Memory - shared memory (primary focus)
  - Shared processors - single instr. multiple data/data parallel (SIMD)
- Parallel processes (threads) need to comm. (speeds cannot be assumed)
  - Message passing & shared mem. - programmer writes explicitly
  - SIMD is parallelized by compiler or written by programmer (CUDA) Programs
    - works well if applicable.
- Shared Memory - all threads see one global memory (programmer view)
  - use loads/stores to access data
  - conceptually similar to uniprocessors + ease of programming (comm. is hard)
  - ↓ latency communicating small data items (often used by real programs)
  - hardware controlled sharing allows automatic data motion
- Message Passing - each process only sees its own private mem (programmer view)
  - uses sends/recieves to access data
  - very simple hardware (almost nothing extra)
  - communication protocol is and must be explicit
  - shared mem. MP can emulate msg passing easily
  - has the highest programming burden, keeping track of comm. artifacts even w/ dynamic comm. patterns
  - have to carefully distr. data to processes
  - partition data or replicate (do repl. in software if not auto. (more work))
  - coalesce small msgs into large msg to scatter to all priv. mems.
- Cannot assume thread relative speeds due to cache & branch preds. stalling all threads on miss.
  - Letting threads slip & slide relative to each other allows for useful work to be done on other threads during miss.

ECE 437 | Chpt 6 cont'd

Multicore cont'd

- Shared Memory – hardware impl.
  - each thread has its own PC, registers, + stack (thread priv. vars are in regs + stack)
  - Rest of memory (heap) is shared (global vars)
  - keep track of num of threads, threadids, + locks.
  - Use macros for creating threads, waiting for threads to finish, barriers to ensure thread synchronization, + lock + unlock to prevent race conditions from introducing issues.
- Message Passing – hardware impl.
  - all variables are private
  - keep track of # of processes
  - use macros for creating processes, waiting for processes, send + recieve msgs.
- SIMD – hardware impl.
  - Shared mem/msg passing are MIMD
  - 1 instr on multiple data, ↓ instr. overhead.
  - Deep, fast pipeline (ex: 50 stages)
  - Parallelism guaranteed by programmer/compiler, no need to check for hazards
  - Works great in some workloads (graphics/matrix mul in AI) but not useful for all applications.
- Parallelization Strategies
  - Best parallel alg isn't always the parallelization of the best seq. alg.
  - Think of partitioning tasks/data
  - Programming Models
    - x Pthreads – shared memory processors
    - x MPI (message passing interface) – clusters
    - x Open MP – shared memory – marking loops as parallel loops
    - x ↑ lvl primitives – streaming, map reduce, parallel recur. (cilk())
      - customized to arch. like CUDA for nvidia GPGPUs.
- Typical multicore
  - private LII, LID, L2 cache, shared banks of L3 cache
  - 64 bit mem. channel + multiple mem. controllers
  - mem + L3 connected to private cache w/ interconnect network.
- "Heavyweight" Process – diff thread of cntrl (PC, reg, stack) + diff addr space (pt, heap)
- "Lightweight" Process – threads – diff thread of cntrl (context) + same addr space
- Can share regions across ↓ processes.

ECE 437 | Chpt 6 cont'd

Multicore (cont'd)

- Cache Coherence - important to maintain shared mem. model
  - hard to track arbitrary sync among arbitrary threads in large distr. sys.
  - so all writes updates all threads, to ensure to prior value isn't readable (write atomicity) without tracting sync.
  - value read must be val. from last write
    - x need to decide how to propogate writes & write serialization (single loc. write) or atomicity.
  - appearance of coherence not true coherency needed (instant updates) not necc.
  - coh. only provides write atomicity (no one can see old val. after write)
    - x writes aren't instantaneous so these rules are followed
      - After writes, nobody can see old value
      - During writes, nobody can see new value
      - Later thread shouldn't see old val. while older thread sees new val.
- Enforcing Coherence.
  - On writes, invalidate or update other copies
  - Use snopping in other caches to ensure other caches are updated and are not hitting on stale data
  - All caches can see the bus - bus transactions to enforce coherence.
    - x cache misses go on bus to snoop in all other caches (looks at tag)
    - x cache controller generates block states on processor/snoop events & generates bus actions.
- Shared Mem Models
  - Shared Cache - processors share a cache which is where all data goes through.
  - Bus-based-shared mem. - covered here
  - Dancehall - similar to bus based but uses an interconnect network & pt to pt connections.
  - Distributed Shared Mem. (DSM) - memory local to each processor w/ an interconnect network.
- Snoopy Design Choices
  - set of states
  - state transition diagram
  - actions

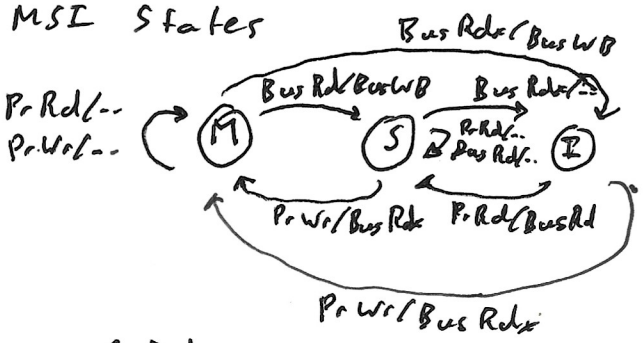
  ] Protocol

  - write-through vs write back
  - update vs invalidate
  - multiple-reader, single-writer

  ] Basic Choices

Multicore cont'd

- 3-State write-back invalidation protocol
  - MSI protocol
    - Modified - stale mem. + one cache has latest valid copy
    - Shared - 1+ caches have valid copy + no stale mem.
    - Invalid
  - invalidate all other copies before M state.
  - use bus tracking to enforce
  - MSI States



[redacted]
(is invalid if no tag match)

  - * PrRd - curr. processor Rd
  - * PrWr - curr. processor Wr.
  - * BusRd - some processor rd.
  - * BusRdx - some processor rd. town.
  - * BusWB - some processor writing back data
  - * A/B = if A, B = A → B
  - Three edge cases
    - * S → M (PrWr/BusRdx) - we have block but mem. still provides blk.
      - need BusRdx for invalidation
      - need to read from mem on BusRdx for I → M.
        - can use seperate Rdxs for I → M & S → M, but its more complicated
      - On cache side, act as cache miss (BusRdx) & PrWr to S.
    - * M → I (BusRdx/BusWB) - bus writeback even though other cache blocks bigger than word, so need to wb entire blk. - processor is overwriting
      - mem sees BusWB + doesn't know if M → S, even though mem will be stale at the end
      - can make sep WB for M → S / M → I, but more complicated
    - * S → S (BusRd/...) - can provide block to read.
      - multiple Ss might exist, leading to need of arbitration + ↑ complexity.

Multicore cont'd

- Details
  - Each cache block has its own state + multiple caches change state (if same blk)
  - Seperate cache controller (per cache), bus controller, + mem. controller for clean & easy impl.
  - BusRd + BusRdx have responses (blk), BusWB has no reply
  - op. sequence:
    1. core req. ld/st
    2. cache hit/miss (cache controller)
    3. bus req. (bus controller)
    4. grant + bus transaction (BusRd/BusRdx on bus)
    5. snoop into other caches (each cache controller)
    6. check MESI state of all caches in parallel (each cache controller)
    7. perform state trans. + update state (each cache controller)
    8. access mem. if needed (mem. controller)
    9. either memory or a cache place block (BusWB) on bus
    10. requestor cache puts blk in cache, updates state, complete ld/st (co.)
    11. data returned to core if needed.
- Bus Controller - maintains coherence + broken into multicycle to maintain fast clk.
  - has seperate state machine from cache controller
  - 5 bus cycles
    1. Bus req. (bus controller)
    2. Arbitrate + bus grant (bus controller)
    3. Snoop + state transit (all cache controllers)
    4. resp. data (wait many cycles if from mem) (cache/memory)
    5. req. compl. + release bus (cache controller)
  - Must latch important signals in the cycles to prevent long delays
    1. latch bus req. (addr, trans...)
    2. latch bus grant (addr, trans,...)
    3. latch snoop resp. signals
    4. latch resp data on bus.
    5. latch resp data in cache.
  - Atomic bus - not necc. (can pipeline) but is way easier.
  - Coh. mem. access chart will be modified to incl. bus controller + multicore.
  - use a duplicate tag array for snoops for no structural hazards between pr + cache/bus snoop.
    - & must update tag array + cache together, but it's ok like tag updates only on eviction/write.
    - & stretch state update till both tags can be updated.

Multicore cont'd

- Coherence makes writes appear atomic
  - hits guaranteed correct data
  - services one miss data line in order received to ensure two parallel write misses
  - completes servicing entirety of one miss before moving on bottlenecking @ bus arbiter.
  - real systems do not use atomic bus (slow)
    - uses split transaction bus (divide req + resp or pipelining)
      - write atomicity is complex - used in small systems
    - uses no bus in distributed shared mem. (DSM)
      - write atomicity still complex - used in modern processors
- Invalidate vs Update
  - Invalidate - one write before many reads
  - Update - many writes before one read.
  - no clear winner, but Update wr. atomicity is hard in DSM systems, so invalidate used in practice, not update
- Synchronization ensures data races don't occur.
  - use locks + mutex to ensure one thread only can run "critical" section at a time.
  - barrier ensures thread sync. before continuing
  - event notification as cross thread signaling
- Atomic R/M/W ops (x is lock var) - op has to be atomic + return prev lockval (r in all cases)
  - Test+set (r, x) (r = m[x], m[x] = 1) r stores lock + checks if = 1 (already locked)
  - Swap (r, x) - swaps r + m[x], r set prev. lockval.
  - Compare + Swap (r1, r2, x) (if r1 = m[x] swap r2, m[x])
    - r2 set prev lockval
  - Fetch + Op (r, x, op) (r = m[x], m[x] = op(m[x])) - r gets prev lockval.
- Lock implementation
  - basic: Lock: while(test(lock_var) == 1);
    Unlock: lock_var = 0;
  - enforcing + doing atomic rmw in hw is hard, so its easier to detect non-atomicity + retry till it works
    - can detect by writing when locking (invalidation of other copies)
  - use lr-sc instead of lw/sw for atomicity
    - lr - load-reserved - stores reservation set addr next to cache
    - sc - store cond. - stores data + inv. other copies if res. not invalidated
      - invalidate then check for consistency. (inv own rs too (just in case))
    - one res. set. per core.
    - no store of sc if res. invalidate, update res accordingly (0)
  - atomicity req. for a lot of OSs

Multicore cont'd

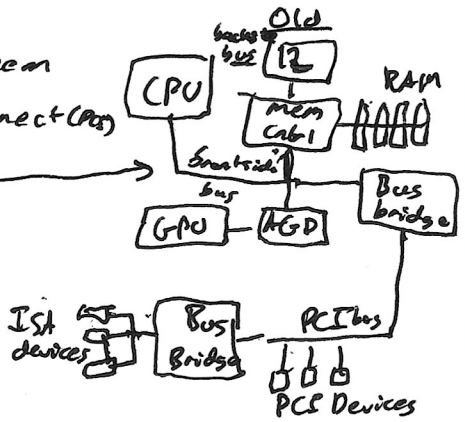
- All sync. stuff is block addr. granularity
  - coherence (inv. is blk sized)
  - v.s. addr. - two locks in same block is bad perf - false sharing but does work
  - If ld ex st done to lock its a software bug.
- Performance
  - T&S is nasty for high contention locks due to constant writes
    - * writing (on all for non-holders,
    - * lot of cache misses due to large # of invalidation
  - can optimize by ↓ t&s exec. + operate + reduce bus traffic.
    - * T&S w/ backoff + insert exp higher delay between t&s
      - not fair as later cond. might get earlier
    - * TTAS keep testing + only t&s when lock is released
      - no cache misses while waiting but still unfair + contention
- Synchronization
  - Deadlocks - hard sw, huge issue, cannot be broken out of w/o external intervention
  - Races - wrong/no lock used leading to racing for data.
  - Timing-dependent - super rare to test/ing and hard to debug
  - Lock composition - hard to divide program into sep. locks.
  - Transactional Mem - not currently ~~[illegible]~~ used + uses blocks of ops in one atomic op.
- Coherence vs Consistency - coh. is for one mem. loc., cons. is for all locs
  - coherent is ops performed by processor appear in prog. order
  - coherent dictates what + consistency dictates when (ordering)
  - reg alloc. in compilation can ruin SC, use volatile to remove reg assignment.
  - ~~[illegible]~~
  - Consistency model is deal between sys. + programmer that tells programmers what behavior to expect.
- Sequential Consistency Model (SC) - Programmers think like this
  - Accesses from one thread in prog. order + writes done atomically.
    - * no rel. speed assumed + each access complete before next starts
  - Seq processors issue mem ops in prog order.
  - Processors take turn randomly to make random # of mem accesses.
    - * gives illusion of sequential thread handling.
  - don't preserve strict mem order for ↑ performance, ↑ software issue w/ parallel prog, other sols could exist.

I/O

- Needed for users, computer, and non-vol. mem. communication.
- Total time = CPU + I/O - overlap
- Transaction Processing does many indep. small I/O ops.
- Performance considerations of I/O depends on applications
- Disks - trends have ↓ $/MB, ↓ disk diameter, ↓ seek time, parallel → serial
  - service time
    - seek time - ~10-20 ms (better w/ locality)
    - rotation time (sector) - ½ rot. on average - 4ms - 8ms (3600-7200 rpm)
    - transfer (xfer) time - pg size transfers ~1-8 kB (several MBs/s)
  - access time = queue time + controller overhead + service time.
- I/O reliability/dependability/availability
  - MTTF - mean time to failure - reliability/availability metric
  - MTTR - mean time to repair.
  - MTBF - mean time between failure - MTBF = MTTF + MTTR
    - availability metric.
  - Bit flips - corrected w/ ECC bits for single/double error detection.
    - done by disk/mem/cache controller
  - Disk errors - easily fixed w/ parity
  - Using banks of data helps error fixing
    - group entire block in one bank + spread different banks.
    - getting first byte is slow so this is better than spreading blocks.
  - Redundant Array of Inexpensive Disks (RAID)
    - provides reliability through redundancy + performance through parallelism.
    - RAID 0 - no redundancy
    - RAID 1 - mirrored for 100% overhead + redundancy
    - RAID 3 - bit interleaving parity - small overhead.
    - RAID 4 - block interleaving parity - small overhead + writes
    - RAID 5 - distr. blockwise parity - small overhead, writes, w/ no bottleneck.
- Connecting I/O to CPU w/ buses. - needed for connecting w/ any I/O.
  - Adv: versatility, low cost, can manage complexity
  - Disad.: comm. bottleneck, speed limited by len + # of devices on bus.
    - need to sup. varying transfer rates + latencies.
  - Processor - Mem bus (design specific) - short, high speed, maximizes data transfers from mem.
  - I/O bus (industry standard) - lengthy + slower, matches variety of I/O + connects to another bus.
  - Backplane bus (standard/not) - interconnection structure for all components.

I/O cont'd

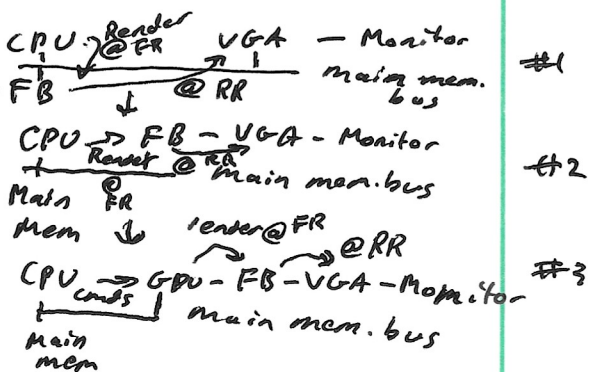
- Bus Architectures
  - one bus (backplane) - all devices on bus
    - + simple + low cost
    - - slow + can cause bottlenecks
  - two bus - uses bus adaptors + I/O buses that interfaces w/ proc-mem bus.
  - three bus - uses a bus adaptor + backplane bus to connect to further bus adaptors + I/O buses to connect to proc-mem bus.
    - x reduces loading on proc. bus.
- Real World Architecture
  - Really old - dual indep bus between proc. + mem
  - Older - Peripheral Component Interconnect (PCI)
    - x 66 MHz sync, 64b data, multi master
    - x AGP used for high speed connection to FSB for GPU
    - x parallel
  - Modern - PCI-e (express)
    - x Serial transmission + faster w/ pt to pt.



- Bus definition
  - Transaction Protocol
  - Timing + Signaling Specification
    - Bunch of wires
  - Electrical Specification
  - Physical/Mechanical Characteristics (connectors)
- Synchronous Bus
  - + little logic + runs fast
  - - must run at same clk for all devices + must not be long if fast to avoid clk skew
  - often uses clk, bus req, bus grant, r/w addr, wait, + data signals
- Async Bus (w/o clk skew)
  - + can be lengthened + accomodate a variety of devices
  - - requires handshaking protocol.
  - often uses req, data, ack, data ready signals
- Flash memory - non-volatile mem, faster than disk, slower than DRAM
  - electronic storage - more reliable, faster + costlier than disk.
  - writes slow bc of wearout (~10k/cell)
  - r/w in blocks.
- Memory-mapped I/O
  - maps I/O devices to VA w/ cntl regs + data buffers/queues
  - I/O is shared w/ only access to OS which sets up VA ⇔ PA mappings.
    - x uncachable for obv. reasons
- I/O devices need to notify OS of when op over/error.
  - Polling - period checks to status reg., simple + only processor ctrled
    - x but overhead = ↑CPU time (↓eff.)
  - I/O interrupts - only halts during transfer, but ↑ interrupt overhead hardware
    - x async
    - x need to comm. status + device
    - x can be handled when convenient, but need to respect interrupt priorities.

I/O cont'd

- Direct Memory Access (DMA) - allows for data transfer from mem. w/o CPU
  - acts as master on bus.
  - uses phys. mem addr. (might be translated by OS)
  - one page per transfer (pinned to prevent pg faults)
- I/O processing
  1. Issue instr to I/O processor (IOP)
  2. IOP looks in mem. for commands
  3. Devices to/from mem transfers are controlled by IOP. (steals mem cycles)
- I/O setup
  - before cache slows CPU, so can't do
  - after cache requires coherence. (applies to DMA too)
  - CPU involvement - low to high involvement
    - Polling - does everything: detection, I/O, copying mem.
    - Interrupts - on notification, does the work
    - DMA - CPU describes work, DMA does it.
    - IOP - CPU tells IOP where to find instrs.
  - OS responsibilities
    - OS interfaces between progs + I/O for protection.
    - provides abstraction, equitable access, handles interrupts, & schedule accesses.
  - Multiprogramming:
    1. Prog. invokes syscall
    2. OS checks prot + runs device drivers
    3. Suspends curr. process + switches process
    4. I/O fielded by OS
    5. OS completes I/O + wakes up suspended process
    6. Next process that is ready is run.
- Video processing
  - Frame buffer - stores frame info (color bytes)
  - Setup #1 is dumb + slows bus
  - using VRAM sep. to main mem (VRAM = video RAM)
    - eliminates structural hazard (#2)
      - dual ported w/ DRAM
      - serial port for Rasterscan w/ shift reg.
  - using independent video accelerator for offloading rendering (GPU)
  - massive reduction + bandwidth without writing fb, issuing commands through graphics APIs
    - APIs like OpenGL + Direct3D



- Other accelerators
  - Per application (ex. GPU for 3D modeling)
  - Broader application (ex. using GPGPU (SIMD parallelization) for computation)

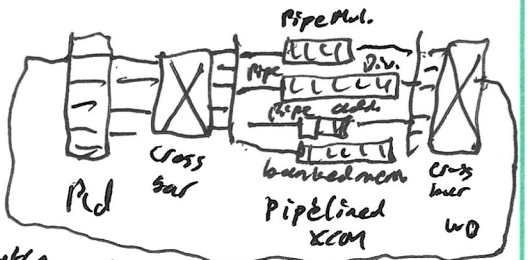
SMT

(SMT)

- Simultaneous Multithreading / Hyperthreading (hardware MT)
  - stalls happen in pipelines even in out-of-order (OOO) processes (cache miss)
  - peak 6-8 IPC, actual 1-2 IPC (25-35% eff)
  - overlap thread execution during stalls
- SMT gives instruction level parallelism
  - run 2-4 threads (or processes) simultaneously on same pipe
  - not context switching in OS but thread switching in software.
  - thread level parallelism - overlapping mem. references to improve throughput.
    - × latency of each thread is same or slower
    - × hardware utilization is better as we overlap cache miss w/ other threads CPU work.
  - 2-4 times more requests to mem but w/ DDR versions, bw will be enough.
    - × does cause capacity pressure on caches.
- SMT implementation
  - need 1 hardware context per thread (multiple instrs in pipe at once)
  - some hw will be duplicated, some will be shared.
  - duplicate:
    - × PC, reg file, wr. buffer, PTBR, reservation set, PID (for correctness)
    - × Br prediction table (for performance)
  - shared:
    - × fetch, decode, ex units, mem, larger caches, TLBs
  - OS views ~~threads~~ contexts as a virtual CPU
    - × assigns processes to each core for all virtual CPUs (threads)
  - SMT allows sharing of most of the pipe for ↑ hardware utilization.
    - × multicore, each core is usually SMT capable.
  - Key decision is where/which context to fetch
    - × rest of pipe doesn't matter other than, reg r/w's, write buffer, PTBR, reservation set, PID.
    - × fetches from non-stalled context + squashes pipe when stalled.
- SMT will be correct memory/caches/TLB as all processes will have diff phys addr. thanks to virt. mem.
  - diff. threads of same process go to same phys. addr.
- Latency
  - L1 & L2 hit (2-4 / 12-15 cycles) - overlapped by OOO in 1 thread
  - L3 hit & mem hit (40-60 / ~300 cycles) - overlapped by OOO in 1 thread & SMT & Multicore (across multiple threads/processes)
  - Mem miss/page fault (millions of cycles) - overlapped by OOO issue in 1 thread & SMT & multicore & OS multitasking (across multiple threads/processes)

Vector

- Used for parallel + regular code like matrix calc in SCI/eng domains
  - CPU made for gen purpose ugly code and thus is complex.
  - CPU complexity makes a seperate vector handling structure more effective.
- Vector has highly parallel regular code and thus wants deep or wide pipe
  - deep = fast clk, wide = ↑parallelism.
  - no data or inner loop hazards (control) so can avoid that h/w
  - multiple ops per instr → statically-sched. in-order wide/deep pipe.
  - uses data level parallelism (DLP) through single instr. multiple data (SIMD)
- High level code → vectors by several processes
  - Compiler w/ automatic vectorization
  - Programmers w/ hand tuned assembly
  - "High level" lang. like CUDA
    - ✗ hard to prog, close to assembly level
    - ✗ huge perf drops if prog. doesn't match h/w
    - ✓ done to optimize tasks really well (like AI)
- For handling vectors wider that SIMD through,
  - pipelined ops in one deep pipe / one wide pipe (multiple func units)
  - use vector regs to store values after retrieving from mem. sub system
    - ✓ also writes back to mem. subsystem. at the end (intermediate stored in regs)
- Vector Mem. system
  - Load/Store from mem into v regs (operates on contig. addr.)
    - ✗ ~~Caches~~ assumes a unit/reg stride length according to vreg/data size
    - ✗ Ex. M[r1 + i*r2]
  - Has lot of spatial locality but not temp. locality. (contiguous streaming access)
  - ↑ compute so ↑ compute per byte
  - system great for high bw DRAM that hides latency w/ prefetching.
    - ✓ heavily banked for ↑ bw
    - ✗ ↑ vregs to hold prefetched data.

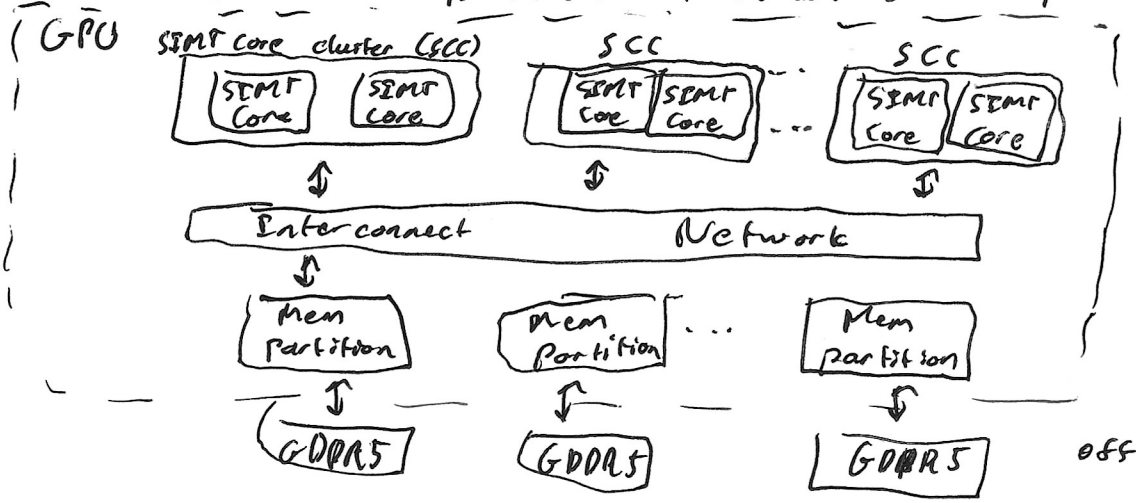


(F|D|Rd|x|m|W is full pipe)

- Strip mining - used if data > vec reg size
  - divides loop into smaller ops of vectors.
  - can programatically set VLEN for ops.
- Indirect Access - use vector indirect ops scatter/gather
  - custom instrs. [ex. M[r1 + v2[i]] → v1[i]]
- Branches - use vector masks
  - hides vector based on cond. (ex. mask[i] = 1 if v1[i] < 50)
- Vector extensions created for Processors
  - ~~Cyphermedi~~
  - commodity vector processors - embedded, DSP, GPU, etc.

GPU

- Started for graphics but GPGPUs now run all kinds of SIMD workloads like ML, Unity, etc.
  - ↑ perf per watt & much more (10x) energy eff than multicore per op.
  - runs a SIMD microarch (diff from normal vector pipes)
  - SIMD processing w/o SIMD instructions
    - x group scalar threads in SIMD exec or SIMT
      - more prog than vectors, but only works if SIMD-amenable
    - x software should be SIMD friendly for good perf.
    - x hw width hidden from software SIMD
    - x hw group = warp in NVIDIA.
- GPUs - graphical processing unit.
  - accerator for raster based graphics (OpenGL, DirectX)
  - ↑ programmable commodity hardware w/ 1000s of ALUs and 10s of 1000s threads concur.
  - exploits TLP + DLP (fundamentally vector processor)
  - uses multithreading to tolerate mem. latency
  - software written for single thread
  - ↑ programmable than vectors (using threads) w/ CUDA
  - SIMT programming - single instr. multi thread.
    - x fine grain threading unlike CPU SMT.
- GPU used for compute bc it uses way more % of silicon for compute compared to CPUs
  - also uses significantly less energy per op.
  - achieved by having minimal cache due to streaming access
    - x ↑ cache recently for ML workloads.
  - runcode w/ warps of lock step threads divided up.



GPU cont'd

- GPU pipelines – wide & deep unlike just deep vector pipes
  - wider for slower clk & energy eff.
  - super simple efficient pipelines w/ 32 lanes in lockstep
    - each lane has reg rd, exe, mem, wb stages
  - no bypass/br pred by hiding latency along w/ mem latency w/ multithreading
  - inorder issue, out-of-order complete.
- Register file – do one wide read for all lockstep threads
  - 4 banks of reg access is enough bc of this
  - read one reg at time when overlap of access
  - only avg case matters as worst case is hidden by MT.
- Branches – cannot use vector masks to keep programmable
  - lock-step lanes assigned if taken / if not taken
  - run is taken first then is not until reconvergence (found by compiler)
  - hardware stack for nested if/else – tracks divergence & convergence
  - too many data dep. branches = ↑ divergence & ↓ perf.
    - graphics often doesn't have data dep branch, so it works.
- GPUs use small L1 cache due to not much reuse in streaming workloads
  - use coalescing of mem access instead of 32 ports to only make 1 block access. (otherwise perf ↓)
- NVIDIA calls each 32-wide pipe as a streaming multiprocessor (SM)
  - each SM has L1 & software managed scratchpad called shared mem. (unrelated)
  - L2 shared by all SMs
  - Mem = global mem.
  - can be 32 wide 32b, 128 wide 8-bit, etc.
  - each chip might have 25–80 SMs
  - lot of access is predictable but not all (texture caches)
- Unpredictable access exposes latency addressed by MT.
  - GPUs have bazillion threads, eg 32 wide pipe w/ 50 way MT.
  - Run any ready context every cycle, schedule w/ scoreboard.
  - High MT possible in GPU due to extremely high parallelism.
    - in programs & data
- GPU mem. – ↑ bw for MT
  - 50 contexts each 32 wide per SM
  - 32 wide usually coalesced access to one blk & many diff contexts on one SM often access same DRAM row (need bw).
  - ↓ locality & ↑ streaming → more banking for bw.
  - can tolerate almost any latency.
  - no coherence (freq. sharing, load+comp, flush sw, low VM sup. in GPU rn.
- No coherence/sharing, or fine grain comm. (not needed)
  - comm. over PCI-e or in integrated GPU.
- Tensor Cores added to GPU for specialized matrix mul.
  - often done in ML & runs way, way faster than SIMD
  - lots of SMs & parallel mul. w/ systolic arrays.