

# EE 5900\_04 Lecture Notes - Spring '01

## EE 5900\_04 Memory System Design & Simulation MWF 13:05-13:55

### Wednesday 1/17- Class # 1

## Introduction

### • Who Am I

- Lecture Style
- From outline - so missing a lecture can seriously disrupt flow
- Please interrupt for questions!

### • My Contact Information

- My Email
  - btdavis@mtu.edu
- All Handouts via Web
  - [http://www.ee.mtu.edu/faculty/btdavis/courses/mtu\\_5900\\_spr01/](http://www.ee.mtu.edu/faculty/btdavis/courses/mtu_5900_spr01/)
- Office Hours - knock

### • Walk thru of Syllabus

- Prereqs!!!! - How many have them?
- Which Prereq?
- PreReq exam on Friday

### • Class movement issues

- Move from Fridays or Mondays to enable 3 day weekends
- ??? Question : Friday 2/2 -> Thursday 2/1 ???

### • Book Issues - 1st vs 2nd Edition

- Use for only the first 1/3 of course
- Suggested Texts have been requested at the library

## Focus & Intent of course

### • Distinction between Computer Engineering & Computer Science

- Computer Science - Efficient use of fixed computer system via software methods
- Computer Engineering - Determination & design of computer system architecture from device specification and requirements

- Co-design - Computer Engineering design task with regard for the range of software tasks to which the hardware will be applied

- **Examination of Computer SYSTEM**

- It is my belief that the primary avenues for increasing computer performance lie in the improvement of memory technologies and the communication mechanisms between components
- Majority of computer engineers do not end up designing processors, but a good fraction do end up on systems
  - Motherboard of desktop workstations <---> microprocessor controlled toaster
  - System design & specification tasks
- How to use available “off-the-shelf” devices to compile the best system for the design specification
  - Using existing bussing techniques - as devices are spec’ed out
  - Using FPGA’s or discrete logic where devices are not available

- **Processor Elements pertaining to Memory Hierarchy**

- Caches & Variants : Associativity, linesize, victim cache, trace cache
- Memory disambiguation
- Load/store operations under speculative executions

- **Memory Technologies & Configurations**

- Many Technologies available
- PROM(Flash) - Easy to read, power intensive to write, slow, small
  - non-volatile ; used for seldom changed programs and/or data
  - EPROM -> EEPROM -> FLASH
- SRAM
  - Widest grouping
  - Register file on mProc, Cache, device buffers
  - volatile
- DRAM
  - Main Memory : SDRAM, PC100, PC133, DDR266, DDR2, DRDRAM
  - volatile & requires refresh

- **Simulation**

- Tradeoffs at the system level are often non-intuitive
  - When extra die area becomes available is it best spent on larger cache, larger tlb, trace cache, victim cache, faster multiplier? - These questions can be answered via simulation
- Simulation can happen at many levels - Spim, Dinero, SimpleScalar, SimOS
- The more accurate your simulation model, the more accurate the results
- A 100% accurate model may be as/more difficult than developing the system

- Simulation Guided research/exploration
  - Use low-accuracy models to focus the area of concentration for more accurate models
  - low-accuracy Architectural level simulations done to examine feasibility
  - Design constraints established & functional blocks partitioned
  - Circuit-level simulations done on functional blocks to verify they meet spec
  - Feedback results into arch-level sim for more accurate results
- We will only be examining a single stage of simulation, as we are not a 100+ person design team

**READING : 4.1-4.4 of H&P by Monday**

## **Friday 1/19 - Class # 2**

### **PreReq Exam - 30 minutes**

## **Monday 1/22 - Class # 3**

- **Reminder of reading assignment**
- **Discuss Lab #1 Assignment**

## **Memory Hierarchy**

- **Pyramid Figure**
  - DRAW DIAGRAM (Pyramid)
  - Reg-File
  - On-Chip Cache
  - Off-Chip Cache
  - DRAM
  - Swap space (Disk)
- **Characteristics**
  - Cost (\$ / bit)
  - Access Latency
  - Quantity (bits / system)
- **8086 - First PC's**
  - DIAGRAM (PC\_XT Arch)
  - Direct Wired to the DRAM - same clock frequency
  - This impacts the ISA's of the era
    - low register counts
    - no prefetch or explicit cache management

- allow for self-modifying code
- Frequency Differential is now 60:1 to 200:1

- **Evolution of Memory Hierarchy**

- When processor  $\lt$  memory freq differentials increased to where it was impacting performance, caches were added
- First on the motherboard - while these cache could maintain freq parity w/ proc
- Then in a multi-level design when only onchip-cache could keep pace w/ proc freq

- **P4 (Willamette) Memory (on-chip) Diagram**

- DIAGRAM (SysOpt webpage)
- Two levels of data cache on-chip
- Two levels of Instruction Cache on-chip
- First level cache is not shared - and is very different
- Third level of cache off chip - dedicated bus

- **System Architectural Diagram**

- DIAGRAM (Via Slot 1 chipset)
- North-Bridge/South-Bridge/AGP
  - Where is memory located in this figure?
  - Many places - just about every component
  - But the processor has the abstraction of a unified memory space?

## Wednesday 1/24 - Class # 4

- BANDWIDTH vs. LATENCY
  - bandwidth - throughput (bytes/sec) of an interconnection
    - Potential - pure calculations (bus-width \* frequency)
    - Effective (or realized) - bandwidth utilized in an interconnection
  - Latency - Time from desire for data/object to retrieval of data/object
    - Many different latencies For Example DRAM latency
    - DRAM advertised latency - from chip boundary to response
      - RAS / CAS / Data retrieval
    - Processor boundary - from access leaving processor to return to processor
      - +++ FSB occupancy ; DRAM Ctrl queueing ; DRAM bus occupancy
    - Processor observed latency - from Core data request to data availability
      - +++ Multiple level miss detection ; synchronization delay @ boundary

# Processor Attributes

## • Pipelining basics (4.1)

- Pipeline CPI = Ideal Pipeline CPI + Structural stalls + RAW stalls + WAR stalls + WAW stalls + Control stalls
  - Goal - to get Pipeline CPI as close to ideal as possible - reduce all other terms to 0
  - Structural stalls  $\rightarrow$  0 as enough resources (transistors) are applied to the problem
  - WAR & WAW stalls  $\rightarrow$  0 when using OO with store buffer
  - Remains : Ideal + RAW stalls (data dependancies) + control stalls (branches)
- Loop Unrolling
  - changing instruction ordering/mix to make more efficient use of functional units and reduce stall cycles
- Data Dependancies
  - EXAMPLE
    - LD            F0, 0(R1)
    - ADDD        F4, F0, F2
    - SD            0(R1), F4
  - Data dependancies cause pipeline stalls (classified in RAW stalls above)
  - Increase CPI
  - Worst case is that a LD has to go all the way out to the DRAM memory system

## Friday 1/26 - Class # 5

### READ : 4-5 through 4.11 for 2/1 - Thursday rescheduled class

- Name Dependancies
  - EXAMPLE
    - LD            F0, 0(R1)
    - ADDD        F4, F0, F2
    - ADDD        F2, F4, F6
    - LD            F6, 0(R2)
  - Can be avoided by register renaming either statically or dynamically
  - Static - Compiler optimized renaming
    - increases register pressure, and is dependant upon ISA - difficult in x86
  - Dynamic - "Register Renaming"
    - increased pipeline depth
- Control Dependancies
  - Subsequent instructions dependant upon preceeding branch

- Contributes to control stalls
- Motivates use of branch prediction (4.3) or speculative execution (

### • **Dynamic Scheduling (4.2)**

- a.k.a out-of-order execution
- not a “new” concept - done in ‘60’s era mainframes
- added to microprocessors when the transistors were available (early ‘90s)
- ?What is your level of understanding? - Not a “single” way to do OO
- DIAGRAM
  - Reservation Stations
  - pipes & RegFile
  - Reorder Buffer
- speedup factor : 1.7-2.5 for CDC6600
- Increases NAME dependancies problems
  - motivates use of a dynamically enamed registers
- ?Why is this critical to understanding the Memory Hierarchy?
  - Without OO techniques, a memory system need only support a single outstanding load
  - With OO techniques, the memory system must support multiple concurrent accesses
  - These techniques also allow the core to continue operation after/during a long-latency memory access - at least until the re-order buffers are full, or all instructions in the reservation stations are data dependant

## **Monday 1/29 - Class # 6**

### • **Speculative Execution**

- Impromptu material

### • **Load-Store Queue**

- Impromptu material

### • **Branch Prediction (4.3)**

- Alot of research has gone into this field - as with highly pipelined designs, branch misprediction significantly impacts performance - Could easily occupy 3 lectures
- Static Branch Prediction
  - Based on “hint” bits encoded into branch instructions
  - Ideal case 2 hint bits - (a) use (b) take/not take
  - Depends upon code analysis by compiler
- Dynamic Branch Prediction
  - Depends upon past history of branches

- one-bit predictor, two-bit predictor, dynamic multi-level predictors

## Wednesday 1/31 - Class # 7

### Reminder - Chapter 4 to have been read by tomorrow

#### • Branch Prediction (4.3) - continued

- Up to 94% accuracy - when allocated enough resources - for SPEC95
- Multiple Level predictors
  - History -> Pattern -> Prediction
  - History can be allocated Globally or per address
  - Amount of history maintained is bits of history
  - Pattern table can be allocated globally or per address
  - number of patterns is determined by the bits of the history register
  - Prediction can be adaptive (2-bit) or static
  - Nomenclature : GAg, GSg, PAg, PAp
- Resources required - increase rapidly, exponentially in some cases
  - When designing a processor, determinations have to be made, where are resources (i.e. transistors or die area) best allocated for optimal performance
  - Storage requirements are significant especially when using ?Ap or PAp?
  - Requirements can be arithmetically determined - but this is not a course on BP
- GShare
  - Use the address of the branch as a partial hash for the history
  - Reduces some of the aliasing, allowing global schemes to approach per-address perf
  - The history is XORed with the history before going to the pattern history table
- Hybrid Predictors
  - Motivated by the realization that 60% of branches have a static outcome
  - Error checking branches, etc - Why let these pollute the dynamic structures
  - MANY hybrid predictors are proposed - two (or more) components predictors chosen by a “meta-predictor”
  - Typically involve a simple two-bit saturating, and a complex two-level
  - Requires still more resources, and complexity, but can increase performance

## Thursday 2/1 - Class # 8 - Rescheduled 3pm in EERC316

- Branch Target Buffer
  - Only accessed by branches “predicted taken”
  - Functions as a predictor of the taken address

- Eliminates the necessity for address generation of branches, which may be difficult due to data dependencies, and certainly requires a cycle
- Eliminates the stall cycles necessary from AG
- Also requires resources
- Return Address Stack
  - RETURNS are Branches with known outcome
  - But target of the branch is not static - thus poorly predicted by BTB
  - Return address stack holds the addresses from which subroutines were invoked
  - allows RET to be 100% predicted w/ 100% accuracy
  - depth of the stack is a parameter, and stack must be able to be pushed to a memory struct

- **SIMD execution**

- impromptu response to questions
- Vector Processors
- ISA Extensions (MMX, SSE, 3DNow!)

## Monday 2/5 - Class #9

- **SuperScalar pipelines (4.4)**

- SuperScalar is essentially putting multiple pipeline in parallel
- ? DIAGRAM - superscalar stairway ?
- ? DIAGRAM - OO superscalar core ?
- Typically not all pipelines are general purpose - and there may be more pipelines than there are available issue slots
- Brings into design parameters a number of new variables
  - Pipeline Types and numbers
    - Parameterizable w/in SimpleScalar
    - ialu, imult, memport, fpalu, fpmult - sometimes branches also - alternatively “branch folding”
    - number of each depends upon instruction mix - fp pipes are larger in die area
  - Fetch Width
    - Determined by bandwidth of connection between L1 cache & memory
    - Even still - frequently limited by alignment issues
    - Variable instruction length (x86) makes this even more difficult
    - Fetch double the bytes required for issue
  - Issue Width
    - Determined by architectural constraints & dependency checking hardware

- Involves identification of instructions in res-stations w/ no unresolved dependancies & transferring from res-stations to pipeline
- This sets Upper bound on “ideal” CPI
- revisit  $CPI = ideal + Structural\ stalls + RAW\ stalls + WAR\ stalls + WAW\ stalls + Control\ stalls$
- Instructions Retired per cycle
  - Upper bound on req'd # is Issue width
  - Actual CPI cooresponds to AVERAGE Instructions retired/cycle over time
  - Involves identification of oldest instructions in reorder buffer - determination if they are complete - then writeback/commit of thier results
  - Easier to impliment than Issue, so less of an architectural constraint
- Pipelines vary in depth & function - best integrated into an OO core

## Wednesday 2/7 - Class #10

**READ : Kroft Paper - "Lockup-Free Instruction Fetch/Prefetch Cache Organization." - By Wednesday 2/14**

**READ : Patel TechReport/Thesis - Pages 1-22 by Wednesday 2/14**

**Please write down questions pertaining to these papers as you are reading them.**

- Limitations to pipeline width
  - dependancy checking grows exponentially
  - Ports to the register files & memory system
  - inherent limitation in ILP
  - difficulty in building devices of that scale
- VLIW - Very long Instruction Word
  - Each “instruction word” contains multiple instructions in the current nomenclature
  - The compiler is responsible for scheduling the dependant instructions such that they can execute without stall
  - Eliminates the dependancy checking within a “word”
  - The EPIC or IA64 architecture is a variant of a VLIW architecture

### • **Load-Store Pipeline(s)**

- One of the required pipeline types in a heterogeneous superscalar core
- Can be a unified L/S pipe - or discrete Load & Store pipes - ADV/DIS
- For x86 instruction set - one of every 5 instructions is a load/store - less for ISA's with more registerers
- A pipeline is required for address generation, and subsequent access stages

- Must have a mechanism for holding Cache-Miss accesses w/o blocking the pipeline
  - MSHR - miss information/status holding register

## Monday 2/12 - Class # 11

- Speculation - Loads can be done speculatively, but Stores can not
  - Store Queue handles this problem
  - Frequently integrated into a load/store queue to eliminate load stalls in the load-pipe(s)
  - Allow stores to be delayed until the commitment of the instruction
  - Allow loads to compare values against preceeding stores
    - Requires Content Addressable Memory - low # of MSHRs/Queue entries
  - Allow loads to stall for high-latency (DRAM, L3 cache) accesses w/o stalling pipeline(s)
  - Entries into the store queue - parameter to the architecture
  - MSHRs are equivalent in functionality to the load/store queues we have been discussing in abstract
- Ports to Memories
  - What is a dual ported memory structure
  - Latency of a dual ported memory structure is higher than that of a single ported
  - This is one advantage to splitting the I/D side L1 caches

### • Trace Cache

- Modern Cache approach - attempts to respond to the Reservation Station FILL problem
- Paper to be read describes implimentation in a RISC context - slightly different usage in a CISC/ hardware emulated context - Examin RISC context first
- Problem : branches & small basic blocks make it difficult to utilize fetch bandwidth
- Solution : cache instructions by basic block & sequence rather than by address
- A sequence of basic blocks in a single trace-cache line is called a segment

## Wednesday 2/14 - Class # 12

- DIAGRAM - Pages 8 & 9 of Patel
- Indexed by fetch address of first basic block & (partially) branch prediction
  - should be associative - more than one block may contain the same first basic block
  - should allow for partial match - if branch predictions are only partially correct
- Trace cache lines are gathered by the Fill Unit
  - Fill Unit is an algorithmic challenge in itself - Patel Suggests 4 rules for segment completion
    1. segment contains 16 instructions
    2. segment contains 3 conditional branches
    3. segment contains an indirect jump, return or trap

- 4. merging the next incoming block would result in segment length > 16
- Implications
  - Instructions may be cached in multiple places in the trace cache
  - The trace cache line must contain target for all branch outcomes - to expedite fetch, and because otherwise basic block addresses must be kept to calculate branch targets - and storage overhead is equivalent.
  - Tag is address (partial if not FA) and branch outcomes
- CISC advantages
  - CISC Decode typically requires conversion from Complex ISA -> Internal ISA
  - True of P3/P4/Athlon - Somewhat true of Crusoe, but not the whole story
  - In this case, the decoded Internal ISA (ROP) instructions can be cached in the trace cache
  - Other advantages - increase fetch effectiveness - are consistent
- Branch Predictor & Instruction Cache
  - These functional units are primarily unchanged from a non Trace-Cache implementation
  - The Branch predictor must be capable of predicting three branches into the future (a separate and complex issue) - generates a set of (3) 2-bit predictors for each branch address
  - The Instruction Cache remains primarily unchanged, but might not require as long a line-size as a non Trace Cache implementation

### • **Compiler Optimizations (4.5) - Not the focus**

- Elimination of Dependancies
  - Dependancies impact performance by restricting the issue of the processor
  - Compilers should attempt to eliminate dependancies where they can be identified
  - loop-carried dependancy - each iteration dependant upon the preceeding iteration - can not be eliminated - but it's
  - register/name dependancy - can be eliminated by compiler techniques if enough registers are available

## **Friday 2/16 - Class # 13**

- Increase parallelism
  - Subtle differences between these techniques
  - Loop Unrolling
    - Unrolling a loop such that multiple iterations of the original loop occur in a single iteration of the unrolled loop
  - Software Pipelining
    - Unrolling a loop such that a single iteration of the original loop is carried across multiple iterations of the unrolled loop - effective as a memory prefetch technique
  - Trace Scheduling

- Critical Path Scheduling & Trace Compaction
- Attacks same problem as trace cache
- Cache line alignment of branch targets / basic blocks
- Makes the frequently executed basic blocks (plural) in a common cache line
- Put infrequently executed branch targets or basic blocks off the fall-through path

• **Hardware Support for Extracting ILP (4.6) - covered somewhat 1/29**

- Techniques requiring support in the: compiler, ISA and hardware
- This means changes to the ISA - can be done by start-from-scratch or ISA additions
- Predicated Execution (rather than branches)
  - Branches have a negative impact upon performance because they change the fetch path
  - Predicated instructions do not change the fetch path, but writeback conditional upon a predicate
  - Many novel architectures include support for predicates (IA64)
  - Common & Simple instructions are frequently predicated, CMOV
  - EXAMPLE
    - if (b!=0) a = a + b ; else a = a + 1
    - ld            r1, 0(r16)
    - pcmp.eq      p1,r1,0
    - add          r0, r0, r1      (p1)
    - addi         r0, r0, 1      (!p1)
  - Increases basic block size - increases fetch bandwidth
  - Book discusses two approaches
    - Predicated instructions w/ predicate explicit - requires many register specifiers and is high latency
    - Poison bits for exception - exceptions can be high latency
    - Most approaches being proposed or implemented now use explicit predicates which can be calculated & used (repeated) in an instruction encoding
- Dynamic Register Renaming
  - Dynamic Register Renaming has been mentioned before - it reduces the pressure on a small register file - or eliminates structural hazards in an OO-execution core
  - Goal : eliminate the structural hazards/stalls caused by name dependencies - with “infinite” register resources these hazards are completely eliminated
  - Allows for multiple versions of a single register NAME depending

**Monday 2/19 - Class # 14**

- May either eliminate completely the physical register file, or may place the physical register file into a background or shadowed role (retirement register file).

- Allows for a different number of “physical registers” from “architected registers”
  - physical registers - number of registers in the register file(s) of the processor, available for use w/o memory access - can be different between implimentation of same ISA
  - architected registers - number of registers available to the instructions - encoded in the ISA
- Requires an additional stage in instruction decode (done IN ORDER) which maps the (architected) instruction encoded register specifiers into (physical) register specifiers - the physical register specifiers then proceed with the instruction through the remainder of the pipeline.
- The state of the register mappings are maintained in the RAT - Register Alias Table - this structure is accessed by the remapping stage, and the ReOrderBuffer - the re-order buffer only needs to access it to determine which physical register to commit to, and how to “roll-back” the RAT in the case of miss-speculation
- EXAMPLE - Dynamic Register Renaming / Name Dependance
  - DIAGRAM - showing DECODE/OO\_CORE/Retirement
  - $r1 = r2 + r3$
  - $r4 = r1 + r5$
  - $r1 = r6 + r7$
  - $r8 = r1 + r4$
  - Architected registers vs. physical registers. Register renaming?
  - Rename the previous example where the Register Alias Table (RAT) is initially:
    - $r1 \rightarrow p12$     $r2 \rightarrow p6$     $r3 \rightarrow p9$     $r4 \rightarrow p15$
    - $r5 \rightarrow p1$     $r6 \rightarrow p10$     $r7 \rightarrow p8$     $r8 \rightarrow p14$
    - Free List:  $p5, p11, p13, p4$ .
- Allows the elimination of all false/Name dependancies (WAR & WAW)
- RAW - “true” dependancies remain
- Explicit prefetch (ISA must support)
  - Allows a known high-latency access to be initiated early, such that other instructions can execute in parallel
  - Can already be done with “load hoisting” why are new instructions necessary
  - Prefetches do not : Pollute registers, cause exceptions
  - Prefetches can be : predicated, predicted address
  - Goal : Eliminates the stalls associated with dependancies upon high-latency loads

## Wednesday 2/21 - Class # 15

### READ : 5.1 - 5.4 by Monday 2/26

- Speculation - Putting it all together

- Branch Prediction - Register Renaming - Reservation Stations - Store Queues - ReOrder Buffers
- DIAGRAM - How these parts fit into an OO superscalar core
- This is the baseline for most modern microprocessors - Not everything is here, but this gives you a good framework for understanding the other portions of the system
- Pipeline's w/in pipelines - Pipelines are a key to faster clock speeds and higher ILP
- Current designs save speculation resolution until the Commit/ROB stage to make recovery easier to implement
- Possible to speculate on multiple contexts - predict down multiple branches
- Possible to contain state for two concurrent threads (SMT) - next generation
  - Maintains highest level of throughput for any one thread
  - Uses extra resources for other threads
- Basic framework for "typical" modern microprocessors - High exploitation of ILP requires

### • The Limits to ILP (4.7)

- ILP - Instruction Level Parallelism
- Assumptions to maximize ILP
  - Register Renaming - infinite registers - all WAR & WAW (name dependencies) are avoided
  - Branch Prediction - perfect branch prediction
  - Jump (Branch Target) prediction - perfect target address prediction
  - Memory disambiguation - all load/store addresses are known and can be resolved OO w/o waiting for address generation
  - Infinite Fetch & Decode bandwidth
- Available ILP : 17-150 for SPEC
- Book goes through constraints on these #'s due to "real" fetch/issue/BP/Registers/Memory Disambiguation
- I am not going to spend time on that - but I do want to address the inherent limits of the way that we encode programs
- Are the limits encoded into our ISA's
- ISA's assume sequential execution of programs
- Sequential execution - regulated by the program counter - of programs makes the "critical path" time through the program longer
- Dataflow - One example which breaks this mode, not the only one
  - DIAGRAM (car assembly?)
  - Focus on the critical path of execution
  - Break workload into component tasks
  - Minimize the dependencies between these tasks

- Execute each of these component tasks upon a unique processor core (may be same chip)
- How do we encode a dataflow program?
- Thread-level-parallelism in Software terminology
- Requires additional annotations to describe multiple starting points or parallel tasks
- Popular in MultiMedia/Graphics engines
- Experimental/Proof-of-concept machines have been built which follow the Dataflow model

## Homework #1 - 4.10 & 4.20 due 2/28

## Memory Technologies

### • Motivation for Virtual Memory - Impromptu

- Use of larger memory space than physically available
- Protection - allows O/S to prevent one process from “mucking with” the data of another process or the O/S
- Relocation - allows O/S to provide each app with it’s own virtual memory space, using whatever virtual addresses the process desire(s)

## Friday 2/23 - Class Cancelled (health)

## Monday 2/26 - Class # 16

### • Impromptu

- Write allocation & write-back/through policies

### • Return to Memory Hierarchy

- DIAGRAM
- What is not shown?
  - Flash BIOS - on MBoard & Controller cards
  - Disk Drive ROM & Caches
  - Graphics Card ROM & Buffers
  - Network Card Buffers
  - Essentially I/O device memories
  - Should be included : Tape, Optical Disk, Magnetic disk, FeCore, Magnetic Drum

### • Memory Characteristics

- Principle of Locality
  - locality of reference : programs tend to reuse data and instructions they have used recently
    - Especially true on stack data types or array processing
    - This is the property which makes long cache lines useful
- Temporal locality

- Items accessed recently in time are likely to be accessed in the near future
- This is the property which makes caches useful
- Spatial locality
  - Items whose addresses are close together are likely to be accessed in close temporal proximity
- Volatile vs. Non-Volatile
  - non-volatile - will retain data even in the absence of supplied power
  - volatile - will retain data only while constraints upon the supply power, and other possible constraints (refresh) are maintained
- Sequential access vs Random Access vs pseudo-random access
  - Sequential access - data locations are read out in address order - changing this order increases the subsequent access times - access time dependant upon distance from last address read
    - Drum, Tape, disk(to some degree)
  - Random access - data locations can be read out in any order - access time is equal for the next byte or a byte 1/2 way across the memory space - completely uniform access time
    - SRAM, \*ROM, Flash, first-generation DRAM
  - Pseudo-random access - non-uniform access time - precise parameters dependant upon specific architecture
    - FPM DRAM, EDO DRAM, SDRAM, DRDRAM
    - note : despite RAM in name - these technologies are not random access

## Wednesday 2/28 - Class # 17

### • SRAM

- Diagram - Logical diagram (printout) - build up gate by gate
  - bistable flip-flop
  - How many transistors to implement?
- Diagram - SRAM Cell (Sharma Fig 2.4) - CMOS(a) & Dual-Port(c)
  - 6-transistor basic cell - higher density than the logical diagram
  - Read Operation - Both Bit &  $\overline{\text{Bit}}$  held high, then enable word
  - Write Operation - Bit &  $\overline{\text{Bit}}$  set to be appropriate for the data to be written, then enable word
- Diagram - SRAM Device (Sharma Fig 2.5)
  - Many “cells” organized in an array
  - Multiple arrays can be organized in “banks”
- Many technologies have been used for Cache design - I have focussed upon CMOS here because it is the most common general purpose process
  - Familiarity w/ NMOS vs CMOS vs PMOS?
  - nMOS - early, fewer transistors, higher power consumption

- CMOS - Complimentary, more transistors, lower power consumption
- BiPolar (DCTL, ECL, BiCMOS) - faster, more power consumption, less mature processes

## Monday 3/12 - Class # 18

- Interface
  - As with most memory devices - while the core can remain constant the interface may vary
  - On chip SRAM can have whatever timing diagrams the designers desire (i.e. Alpha dual-speed on-chip cache)
  - Many varieties of SRAM are available (asynch, synch, synch-burst, sequential)
  - Asynch Read Timing Diagram (TI printout) - Diagram & Block Diagram
  - Flow-through Read Timing Diagram (TI printout) - Diagram & Block Diagram
  - Piplelined Read Timing Diagram (TI printout) - Digarm & Block Diagram
  - SRAM Write Timings
    - Default(TI printout)- why is this bad?
    - Late Write & Zero Bus Turnaround (TI printout) - why better?
- SRAM placement
  - How is SRAM different than say a register file - not - 32 bit lines w/ many ports
  - Mproc structures - Branch Target Buffers, caches, reservation stations, ROB all implimented as SRAM
  - Frequently have “sub-block” writes, multi-port, or other use-specific designs

## Wednesday 3/14 - Class # 19

### • Caches

- Caches can be built of any memory structure - but are almost always SRAM
- Cache line & cache block are typically synonymous
- Caches by definition cover only a portion of the entire memory space
- This means that they require TAGS to determine which portions of the memory spare are covered
- Where can a block be placed in a cache
  - Anywhere - Fully Associative
  - A single location - Direct Mapped (1-way set associative)
  - N possile locations (N-way set associative)
- How is a block found in the cache
  - Physical Address components
    - byte-in-block (block offset) - low order
    - index - set of bits from above byte-in-block (not necessarily low-order or contiguous)
    - tag - remaining bits

- Typical for index to be low order, but not necessary
- Retrieve all blocks in set & compare tags in parallel
- Block allocation/Replacement
  - When a set is completely populated, yet a new block must be brought into the set, how do we determine which block is ejected
  - clean first- replace clean blocks in the set first, minimizing bus traffic - does it? - can be integrated with any of the policies listed below
  - LRU - least recently used - most typically implemented as “pseudo lru”
  - FIFO - first in first out
  - random - randomly select a block to be ejected
  - Ideal - longest latency to next use
  - Policy decision - all will work correctly, seeking the policy yielding the best performance. May be different for different benchmarks/workloads
- Write Policy
  - Covered w/ regard to write allocation on 2/26
  - Write Through - every write goes through to the lower level cache, thus a line does not require a write when ejection occurs
  - Write Back - dirty bytes/blocks are maintained in the cache - requiring dirty bit(s) for the cache block or sub-blocks. These dirty lines must then be written back upon ejection
  - Write Back vs. Write Through - Write back yields lower traffic on the bus below the cache being discussed, but can yield higher latencies when a block must be written back prior to retrieval of a freshly allocated block
  - Write Allocation - When a write occurs to a line not presently in the cache, the cache designers can choose to allocate that line into the cache, or pass the write down the memory hierarchy to be written at that level.
  - Cache coherence - in SMP environments, cache coherence requires that processors be aware of what (dirty) cache lines are being maintained by their peer processors. This allows the processors to maintain a consistent state of the shared memory. One protocol which is designed for this purpose is the MESI protocol.

## • Cache Misses

- The three C's of Cache misses
- Compulsory - cold start misses - first access to block, so block must be retrieved - can be reduced by larger cache lines or via prefetching
- Capacity - block desired was previously in the cache, but ejected due to cache capacity limitations - can be reduced by increasing cache size
- Conflict - block desired was previously in the cache, but ejected due to capacity limitations within the set of this block - can be reduced by increasing cache associativity

## **End of Material for Exam # 1**