

G22.2243-001
High Performance Computer Architecture

Lecture 9
 Memory Hierarchy Design (cont'd)

March 22, 2006

Outline

- Announcements
 - Lab Assignment 3 due back next week March 29th
 - Assignment 4 will be available that day
- Memory hierarchy design
 - Cache optimizations
 - Main memory

[Hennessy/Patterson CA:AQA (3rd Edition): Chapter 5]

3/22/2006 2

Recap: Improving Cache Performance

CPU time = (CPU execution clock cycles + **Memory stall clock cycles**) x clock cycle time

Memory stall clock cycles = (Reads x Read miss rate x Read miss penalty +
 Writes x Write miss rate x Write miss penalty)
 = Memory accesses x **Miss rate x Miss penalty**

- Above assumes 1-cycle to hit in cache
 - Hard to achieve in current-day processors (faster clocks, larger caches)
 - More reasonable to also include hit time in the performance equation

Average memory access time = **Hit Time + Miss rate x Miss Penalty**

Small/simple caches Avoiding address translation Pipelined cache access Trace caches (D)	Larger block size Larger cache size Higher associativity Way prediction Compiler optimizations (B)	Multilevel caches Critical word first Read miss before write miss Merging write buffers Victim caches (A)	Nonlocking caches Hardware prefetching Compiler prefetching (C)
---	--	---	---

3/22/2006 3

Recap: Cache Optimization Summary

Technique	MP	MR	HT	Complexity
Multilevel caches	+			2
Early Restart & Critical Word 1st	+			2
Priority to Read Misses	+			1
Merging write buffer	+			1
Victim Caches	+	+		2
Larger Block Size	-	+		0
Higher Associativity		+	-	1
Pseudo-Associative Caches		+		2
Compiler Reduce Misses		+		0
Non-Blocking Caches	+			3
HW Prefetching of Instr/Data	+	+		2/3
Compiler Controlled Prefetching	+	+		3
Avoiding Address Translation			+	2
Trace Cache			+	3

3/22/2006 4

Static Random Access Memory (SRAM)

- A type of semiconductor memory used for **Caches**
- Memory retains its contents as long as power remains applied → **Static** unlike dynamic RAM (DRAM) that needs to be periodically refreshed
- Nevertheless it is volatile memory
- **Six transistors** for each memory cell (bit)
- Gate level access time

3/22/2006 5

Dynamic RAM (DRAM)

- Uses smaller number of transistors (**one transistor** per cell)
- Uses a **capacitor**
- Changes in the charge are detected and amplified

- Cheaper than SRAM
- Slower than SRAM

3/22/2006 6

DRAM (Cont'd)

- Reading is destructive
 - When a bit is read you destroy the stored value (if it was a zero)
 - After reading a bit, you must **re-write** it (if it was a zero - in practice it is easier to re-write the value in all cases)
- Stored values must be periodically refreshed
 - Bits are stored using capacitance, it is necessary to periodically re-write (**refresh**) the stored values because charge leaks away over time → **dynamic**
 - Refresh is typically managed by the memory subsystem
- Reading is a relatively slow process
 - Because of need to re-write the contents of a bit that has been read, a **stabilization** period is required after reading
 - Process of detecting changes via sense amplifiers is quite **slow**

3/22/2006

7

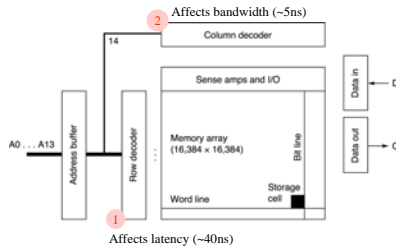
Cache vs. Main Memory

- **Cache uses SRAM:** Static Random Access Memory
 - No refresh (6 transistors/bit vs. 1 transistor/bit, more wires, area is 10X)
 - Address not divided
- **Main Memory is DRAM:** Dynamic Random Access Memory
 - Dynamic since needs to be refreshed periodically (~8 ms, <5% time)
 - Addresses divided into 2 parts (Memory as a 2D matrix) sent one at a time to reduce the number of address pins:
 - **RAS** or Row Access Strobe, **CAS** or Column Access Strobe
- Performance of Main Memory:
 - **Latency**
 - **Access Time:** time between request and when word arrives
 - **Cycle Time:** time between requests
 - **Bandwidth**
- DRAM/SRAM **size** ratio of 4 – 8 for comparable technologies, **SRAM/DRAM cost, cycle time** ratio 8 – 16

3/22/2006

8

Internal Organization of a 256M bit DRAM



- Internally, might use **banks** of memory arrays
 - E.g., 256 1024x1024 arrays, or 64 2048x2048 arrays
- Normally packaged as **dual inline memory modules (DIMMs)**
 - Typically 4-16 DRAM chips, 8 byte wide

3/22/2006

9

Improving Memory Performance in a DRAM

- Increasingly important because fewer chips/system

Evolutionary

- **Fast page mode**
 - Allow multiple CAS accesses without need for intervening RAS
 - Optimizes sequential access, exploiting the **row buffer** (1024-2048 bits)
 - **Extended Data Out (EDO):** 30% faster in page mode
 - Overlaps data output with CAS toggling
- **Synchronous DRAM (SDRAM)**
 - Avoid need for handshaking between chip and memory controller
 - Chip also has a register with number of requested bytes: these are transmitted without explicit requests from controller
- **Double Data Rate (DDR) DRAM**
 - Transmit data from chip on both the **falling** and **rising** edge of clock signal
 - **DDR2** is the next-generation DDR memory technology which features faster speeds, higher data bandwidths, lower power consumption, and enhanced thermal performance

3/22/2006

10

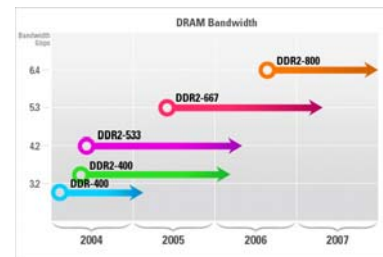
DRAM History

- DRAMs: capacity **+60%/yr**, cost **-30%/yr**
 - 2.5X cells/area, 1.5X die size in ~3 years
- Rely on increasing numbers of computers and memory per computer
 - SIMM or DIMM is replaceable unit
 - computers can use any generation DRAM
 - **Growth slowing because demand is coming down**
- Commodity industry
 - High volume, low profit, conservative
 - Little organization innovation in 20 years
- Order of importance: (primary) **Cost/bit**, (secondary) **Capacity**
 - First RAMBUS: 10X BW, +30% cost, but little impact

3/22/2006

11

Higher Bandwidths



3/22/2006

12

Error Correction

- Motivation:
 - Failures/time **proportional** to number of bits!
 - As DRAM cells shrink, more vulnerable
- Went through period in which failure rate was low enough without error correction that people didn't do correction
 - DRAM banks too large now
 - Servers always corrected memory systems
- Basic idea: add redundancy through parity bits
 - Simple but wasteful version:
 - Keep three copies of everything, vote to find right value
 - 200% overhead
 - ECC (error correction code) SDRAM is memory that is able to detect and correct some SDRAM errors
 - Replaced parity memory which could only detect, but not correct errors
 - Most ECC SDRAMs can correct single bit errors and detect, but not correct larger errors
 - One example: 64 data bits + 8 parity bits (11% overhead)

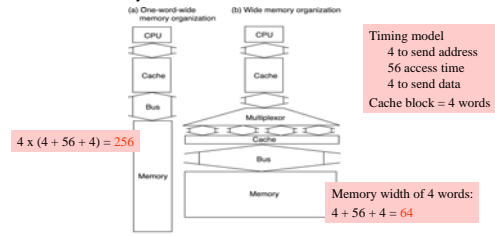
3/22/2006

13

Improving Main Memory Performance

- Making memory faster has been difficult
- At least try to get it to transfer a **lot** of data → higher memory bandwidth

1) Wider Main Memory:



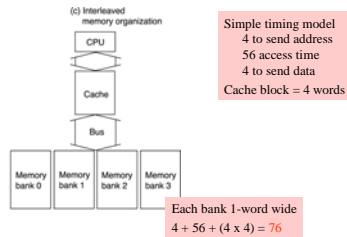
Require multiplexor, wider bus
Difficulties for writes to portion of blocks when using error detection
3/22/2006

14

Improving Main Memory Performance (Cont'd)

2) Use interleaved memory

- Assume four banks are interleaved at word level



15

Improving Main Memory Performance (Cont'd)

3) Generalization: Independent Memory Banks

- Memory banks for independent accesses vs. faster sequential accesses
 - Multiprocessors
 - I/O
 - CPU with Hit under n Misses, Non-blocking Caches
 - Each bank needs separate address and possibly data lines

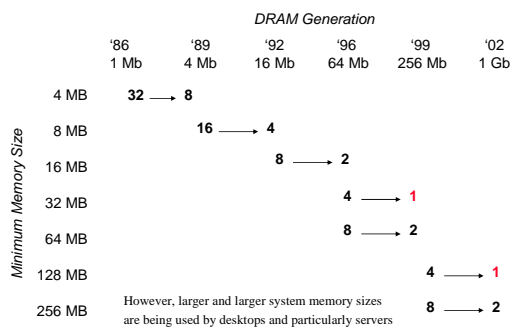
New terminology

- Memory is organized as **Superbanks** of possibly word-interleaved banks
- Each superbank has separate address and possibly data lines
- How many banks?
 - Ensure that if memory is being accessed sequentially (e.g. when processing an array) then by the time you try to read a second word from a bank, **the first access has finished**
 - Unfortunately, **larger memory chips implies fewer banks**

3/22/2006

16

DRAMs per PC over Time



3/22/2006

17

Avoiding Bank Conflicts

- Even if we assume that there are lots of banks, run into **conflicts**

```
int x[256][512];
for (j = 0; j < 512; j = j+1)
  for (i = 0; i < 256; i = i+1)
    x[i][j] = 2 * x[i][j];
```
- With 128 banks, conflict on word accesses (512 is a multiple of 128)
- Software fixes: loop interchange, or **padding** array so that it is not 2^k
- Hardware fix: **Prime** number of banks, **b**, each with **n** words
 - Property:** No conflicts for any sequence of consecutive addresses, as long as stride is not a multiple of **b**
 - Problem:** Resolving the address to a bank number, address within bank
 - bank number = address mod **b**
 - address within bank = address / **b**
 - modulo and divide per memory access are easy if number of banks is 2^k
 - for prime number of banks, harder (particularly /)

3/22/2006

18

Address Computation w/ Prime Number of Banks

- Fast computation is possible by storing words in banks using **modulo interleaving** (b banks, $n = 2^c$ words per bank) ...
 - bank number = address mod b (same as before)
 - address within bank = address mod 2^c
- Above result stems from the **Chinese Remainder Theorem**

Bank Number:	Seq. Interleaved			Modulo Interleaved		
	0	1	2	0	1	2
Address within Bank:	0	1	2	0	16	8
0	0	1	2	0	1	17
1	3	4	5	9	10	2
2	6	7	8	18	19	11
3	9	10	11	3	4	20
4	12	13	14	12	13	5
5	15	16	17	21	22	14
6	18	19	20	6	7	23
7	21	22	23	15	16	23

3/22/2006

19

Lab Assignment 3

3/22/2006

20

A Quick Look

- Simulator sim-multifu:
 - In-order issue
 - Out-of-order execution
 - In-order commit
- Pipeline stages:
 - Fetch (IF)
 - Dispatch (ID1)
 - Issue (ID2)
 - EXE (2nd cycle)
 - EXE (3rd cycle)
 - ...
 - Writeback (WB)
 - Commit (CM)

3/22/2006

21

Multi-cycle Functional Units (FUs)

- Multi cycle functional units (FUs)
 - Operation latency
 - Cycles until result is ready for use
(ready for WB in "current cycle + operation latency" cycle)
 - Issue latency
 - number of cycles before another operation can be issued on this resource
(ready for Issue in "current cycle + issue latency" cycle)

```
char *name;          /* name of functional unit */
int quantity;       /* total instances of this unit */
int busy;           /* non-zero if this unit is busy */
int class;          /* matching resource class */
int oplat;          /* operation latency: cycles until result is ready for use */
int issuelat;       /* issue latency: number of cycles before another operation can be issued on this resource */
```

```
"integer-ALU", 1, 0, { { IntALU, 1, 1 } }
"integer-MULT/DIV", 1, 0, { { IntMULT, 3, 1 }, { IntDIV, 20, 19 } }
"memory-port", 1, 0, { { RdPort, 1, 1 }, { WrPort, 1, 1 } }
"FP-adder", 1, 0, { { FloatADD, 2, 1 }, { FloatCMP, 2, 1 }, { FloatCVT, 2, 1 } }
"FP-MULT/DIV", 1, 0, { { FloatMULT, 4, 1 }, { FloatDIV, 12, 12 }, { FloatSQRT, 24, 24 } }
```

3/22/2006

22

Reservation Stations

```
struct reservation_station {
    /* instruction info */
    md_inst_t IR;
    enum md_opcode op;
    md_addr_t PC;
    md_addr_t addr; /* effective address for LD/STs */
    int will_exit;
    /* RS info */
    INST_TAG_TYPE tag; /* reservation station tag; increment to invalidate RS */
    INST_SEQ_TYPE seq; /* instruction sequence, used to sort the ready list and tag
    instruction */
    int in_LSQ; /* non-zero if op is in LSQ */
    int ea_comp; /* non-zero if op is an addr comp */
    /* instruction status */
    int queued; /* operands ready and queued */
    int issued; /* operation is/was executing */
    int completed; /* operation has completed execution */
    /* output dependent links */
    int onames[MAX_ODEPS]; /* output logical names */
    int odep_ready[MAX_ODEPS]; /* output operand ready? */
    /* input dependent links */
    int inames[MAX_IDEPS]; /* input logical names */
    int idep_ready[MAX_IDEPS]; /* input operand ready? */
}

```

3/22/2006

23

Circular Queues

Register Update Unit (RUU): combination of reservation stations and reorder buffer device, organized as a **circular queue** *

```
static struct reservation_station *RUU; /* register update unit */
static int RUU_head, RUU_tail; /* RUU head and tail */
static int RUU_num; /* num entries in RUU */
```

Load/Store Queue (LSQ): holds loads and stores in program order, indicating status of load/store accesses (will be used for dynamic memory disambiguation in Assignment 4) *

```
static struct reservation_station *LSQ; /* load/store queue */
static int LSQ_head, LSQ_tail; /* LSQ head and tail */
static int LSQ_num; /* num entries in LSQ */
```

3/22/2006

24

Registers

/* Create Vector maps a logical register to a creator in the RUU (and specific output operand) or the architected register file (if RS_link is null) */

```
struct CV_link {
    struct reservation_station *rs; /* creator's RS */
    int odep_num; /* specific operand/register num */
};
static struct CV_link CVLINK_NULL = {NULL,0};
static struct CV_link create_vector[MD_TOTAL_REGS];
```

3/22/2006

25

Referring to Reservation Stations

/* reservation station link: each RS_LINK node contains a pointer to the reservation_station entry it references along with an instance tag */

```
struct RS_link {
    struct RS_link *next;
    struct reservation_station *rs; /* referenced RS */
    INST_TAG_TYPE tag; /* instr tag */
    union {
        tick_t when; /* time stamp of entry (for eventq) */
        tick_t when; /* time when entry will be done (for eventq) */
        INST_SEQ_TYPE seq; /* instr seq no. (for readyq) */
    } x;
};
```

3/22/2006

26

More Queues

/* Ready Instruction Queue contains instructions that have all of their *register* dependencies satisfied */

```
static struct RS_link *ready_queue;
```

/* pending event queue, sorted from soonest to latest event (in time): contains RS_link entries for instructions that are in execution */

```
static struct RS_link *event_queue;
```

3/22/2006

27

Fetch (IF)

- Fetch instructions and put them in Fetch-Dispatch queue

3/22/2006

28

Dispatch (ID1)

- **Note that instruction are sent to issue in order.**
- In each cycle, if the top instruction cannot be sent to the issue stage keep it as last instruction to work on
- In each cycle see if there is a "last instruction" and deal with that; if not get a new one
- Perform functional simulation
- For mispredicted branches squash all instructions in the IF/ID queue
 - fetch_head = (FETCH_QUEUE_SIZE - 1); fetch_num = 1; fetch_tail = 0;
 - Assuming that at the end of dipatch fetch_num is decremented and fetch_head is adjusted
- Create RUU and LSQ entries if need be
- Read operands
 1. If no dependencies, then put it in ready queue
 2. At the same time set the create_vector indicating you will produce the value of a register if need be

3/22/2006

29

Dispatch (Cont'd)

- **Loads and stores are handled in a special way**
 - Split the operation into two ops:
 - An add instruction for calculating the effective address installed in RUU
 - An LD/ST operation installed in LSQ
 - Until the LD/ST effective address calculation is done and the LD/ST is put in ready queue, no other instruction will be issued

3/22/2006

30

Issue (ID2)

- Assign the top of ready queue to a FU
- Schedule events such that we know when it will be done

```
fu = rs_get( fu_pool, MD_OP_FUCLASS(rs->op) );
if ( fu )
{
    /* can issue instruction */
    readyq_pop();
    rs->issued = TRUE;
    /* schedule when next instruction can be issued to this FU */
    fu->master->busy = fu->issuelat; // WILL BE RESET IN WB ( release_fu )
    /* schedule a result writeback event */
    eventq_queue_event( rs, sim_num_cycles + fu->oplat );
}
}
```

3/22/2006

31

Issue (Cont'd)

- Issue of "store" is different; Marked as "completed" in the same cycle
- No entry in events queue (eventq)

```
if ( rs->in_LSQ &&
    ((MD_OP_FLAGS(rs->op) & (F_MEMF_STORE)) == (F_MEMF_STORE)) ) {
    /* instruction is a store */
    .....
    if ( verbose ) print_verbose_message( "write", rs );
}
}
```

- Does not consume WB resources; if previous instruction also in WB at the same cycle, there will be two in WB in same cycle. **The output of pipe2 will only show one.**

3/22/2006

32

Execute (EX)

- Using op latency of one:
 - IF ID IS WB CM
 - IF ID IS WB CM
- Using op latency of two:
 - IF ID IS EX WB CM
- Using op latency of three:
 - IF ID IS EX EX WB CM

3/22/2006

33

Write Back (WB)

- release_fu();
- writeback();
- Check event queue if any instruction is done mark as complete and update the create vector
- **Note that FU is released even though the instruction may get stuck in WB** (because WB is being used by another instruction)
 - In other words, as soon as an instruction spends "operation latency" number of cycles in ISSUE and EXEs, corresponding FU becomes available
- Set the output registers as available (create_vector) here
 - this means in the same cycle a register is written to, another instruction Waiting for it can finish its Dispatch and get into ISSUE in the following cycle

```
IF ID IS WB CM      producer
IF ID IS WB CM      consumer
```

3/22/2006

34

Commit

- Examine head of RRU queue until reaching an incomplete inst
- For these instructions
 - If store and there is port available do it and commit
 - If everything goes as planned release RUU and LSQ (if need be)
 - As soon as a problem (e.g. store cannot complete) stop for this cycle

3/22/2006

35