ECE/CS 552: Review for Final

Instructor:Mikko H Lipasti

Fall 2010
University of Wisconsin-Madison

Midterm 2 Details

- Final exam slot: Mon., 12/20, 12:25pm, EH2317
- No calculators, electronic devices
- Bring cheat sheet
  - 8.5x11 sheet of paper
- Similar to midterm
  - Some design problems
  - Some analysis problems
  - Some multiple-choice problems
- Check learn@uw for recorded grades

2

Midterm Scope

- Chapter 3.3-3.5:
  - Multiplication, Division, Floating Point
- Chapter 4.10-4.11: Enhancing performance
  - Superscalar lecture notes
  - MIPS R10K reading on course web page
- Chapter 5: Memory Hierarchy
  - Caches, virtual memory
  - SECDED (handout)
- Chapter 6: I/O
- Chapter 5.7-5.9, 7: Multiprocessors
  - Lecture notes on power and multicore
  - Lecture notes on multithreading

3

Integer Multiply and Divide

- Integer multiply
  - Combinational
  - Multicycle
  - Booth's algorithm
- Integer divide
  - Multicycle restoring
  - Non-restoring

4

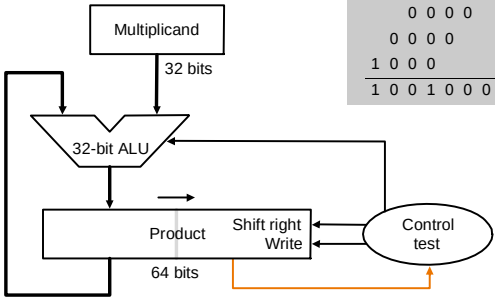
Multiplication

- Flashback to 3rd grade
  - Multiplier
  - Multiplicand
  - Partial products
  - Final sum
- Base 10: 8 x 9 = 72
  - PP: 8 + 0 + 0 + 64 = 72
- How wide is the result?
  - log(n x m) = log(n) + log(m)
  - 32b x 32b = 64b result

			1	0	0	0	
		x	1	0	0	1	
			1	0	0	0	
		0	0	0	0		
	0	0	0	0			
1	0	0	0				
1	0	0	1	0	0	0	

5

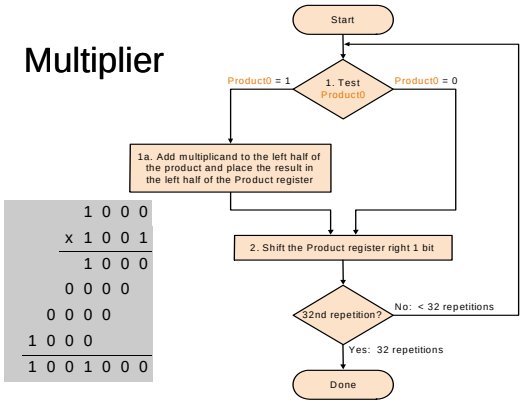
Multiplier



			1	0	0	0	
		x	1	0	0	1	
			1	0	0	0	
		0	0	0	0		
	0	0	0	0			
1	0	0	0				
1	0	0	1	0	0	0	

6

Multiplier




```
      1 0 0 0
    x 1 0 0 1
    ---------
      1 0 0 0
    0 0 0 0
  0 0 0 0
1 0 0 0
---------------
1 0 0 1 0 0 0
```


Booth's Algorithm

Current bit	Bit to right	Explanation	Example	Operation
1	0	Begins run of '1'	0000111**10**00	Subtract
1	1	Middle of run of '1'	0000**11**1000	Nothing
0	1	End of a run of '1'	000**01**111000	Add
0	0	Middle of a run of '0'	0**00**01111000	Nothing

Booth's Encoding

- Really just a new way to encode numbers
  - Normally positionally weighted as 2^n
  - With Booth, each position has a sign bit
  - Can be extended to multiple bits

0	1	1	0	Binary
+1	0	-1	0	1-bit Booth
+2		-2		2-bit Booth

2-bits/cycle Booth Multiplier

- For every pair of multiplier bits
  - If Booth's encoding is '-2'
    - Shift multiplicand left by 1, then subtract
  - If Booth's encoding is '-1'
    - Subtract
  - If Booth's encoding is '0'
    - Do nothing
  - If Booth's encoding is '1'
    - Add
  - If Booth's encoding is '2'
    - Shift multiplicand left by 1, then add

2 bits/cycle Booth's

1 bit	Booth
00	+0
01	+M;
10	-M;
11	+0

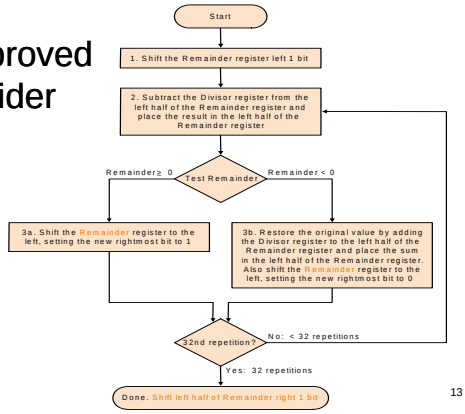
Current	Previous	Operation	Explanation
00	0	+0;shift 2	[00] => +0, [00] => +0; 2x(+0)+(+0)=+0
00	1	+M; shift 2	[00] => +0, [01] => +M; 2x(+0)+(+M)=+M
01	0	+M; shift 2	[01] => +M, [10] => -M; 2x(+M)+(-M)=+M
01	1	+2M; shift 2	[01] => +M, [11] => +0; 2x(+M)+(+0)=+2M
10	0	-2M; shift 2	[10] => -M, [00] => +0; 2x(-M)+(+0)=-2M
10	1	-M; shift 2	[10] => -M, [01] => +M; 2x(-M)+(+M)=-M
11	0	-M; shift 2	[11] => +0, [10] => -M; 2x(+0)+(-M)=-M
11	1	+0; shift 2	[11] => +0, [11] => +0; 2x(+0)+(+0)=+0

Integer Division

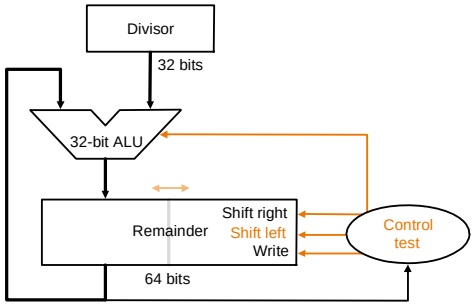
- Again, back to 3rd grade

```
                    1 0 0 1   Quotient
Divisor  1 0 0 0 | 1 0 0 1 0 1 0   Dividend
               -   1 0 0 0
                    ---------
                        1 0
                        1 0 1
                        1 0 1 0
                    -   1 0 0 0
                        ---------
                        1 0   Remainder
```

Improved Divider



Improved Divider



Non-restoring Division

- Consider remainder to be restored:
  $R_i = R_{i-1} - d < 0$
  - Since R_i is negative, we must restore it, right?
  - Well, maybe not. Consider next step i+1:
    $R_{i+1} = 2 \times (R_i) - d = 2 \times (R_i - d) + d$
- Hence, we can compute R_{i+1} by not restoring R_i, and adding d instead of subtracting d
  - Same value for R_{i+1} results
- Throughput of 1 bit per cycle

NR Division Example

Iteration	Step	Divisor	Remainder
0	Initial values	0010	0000 0111
	Shift rem left 1	0010	0000 1110
1	2: Rem = Rem - Div	0010	1110 1110
	3b: Rem < 0 (add next), sll 0	0010	1101 1100
2	2: Rem = Rem + Div	0010	1111 1100
	3b: Rem < 0 (add next), sll 0	0010	1111 1000
3	2: Rem = Rem + Div	0010	0001 1000
	3a: Rem > 0 (sub next), sll 1	0010	0011 0001
4	Rem = Rem – Div	0010	0001 0001
	Rem > 0 (sub next), sll 1	0010	0010 0011
	Shift Rem right by 1	0010	0001 0011

Floating Point Summary

- Floating point representation
  - Normalization
  - Overflow, underflow
  - Rounding
- Floating point add
- Floating point multiply

Floating Point

- Still use a fixed number of bits
  - Sign bit S, exponent E, significand F
  - Value: $(-1)^S \times F \times 2^E$
- IEEE 754 standard 

	Size	Exponent	Significand	Range
Single precision	32b	8b	23b	$2x10^{+/-38}$
Double precision	64b	11b	52b	$2x10^{+/-308}$

Floating Point Normalization

- S,E,F representation allows more than one representation for a particular value, e.g.
  $1.0 \times 10^5 = 0.1 \times 10^6 = 10.0 \times 10^4$
  - This makes comparison operations difficult
  - Prefer to have a single representation
- Hence, normalize by convention:
  - Only one digit to the left of the floating point
  - In binary, that digit must be a 1
    - Since leading '1' is implicit, no need to store it
    - Hence, obtain one extra bit of precision for free

19

FP Overflow/Underflow

- FP Overflow
  - Analogous to integer overflow
  - Result is too big to represent
  - Means exponent is too big
- FP Underflow
  - Result is too small to represent
  - Means exponent is too small (too negative)
- Both raise an exception under IEEE754

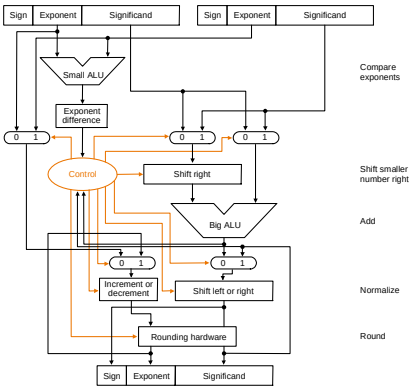
20

FP Rounding

- Rounding is important
  - Small errors accumulate over billions of ops
- FP rounding hardware helps
  - Compute extra guard bit beyond 23/52 bits
  - Further, compute additional round bit beyond that
    - Multiply may result in leading 0 bit, normalize shifts guard bit into product, leaving round bit for rounding
  - Finally, keep sticky bit that is set whenever '1' bits are "lost" to the right
    - Differentiates between 0.5 and 0.500000000001

21

FP Adder



22

FP Multiplication

- Sign: $P_s = A_s$ xor B_s
- Exponent: $P_E = A_E + B_E$
  - Due to bias/excess, must subtract bias
    $e = e1 + e2$
    $E = e + 1023 = e1 + e2 + 1023$
    $E = (E1 - 1023) + (E2 - 1023) + 1023$
    $E = E1 + E2 - 1023$
- Significand: $P_F = A_F \times B_F$
  - Standard integer multiply (23b or 52b + g/r/s bits)
  - Use Wallace tree of CSAs to sum partial products

23

FP Multiplication

- Compute sign, exponent, significand
- Normalize
  - Shift left, right by 1
- Check for overflow, underflow
- Round
- Normalize again (if necessary)

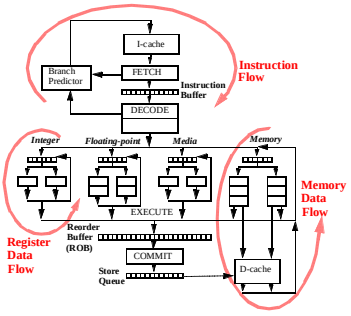
24

Limitations of Scalar Pipelines

- Scalar upper bound on throughput
  - IPC <= 1 or CPI >= 1
  - Solution: wide (superscalar) pipeline
- Inefficient unified pipeline
  - Long latency for each instruction
  - Solution: diversified, specialized pipelines
- Rigid pipeline stall policy
  - One stalled instruction stalls all newer instructions
  - Solution: Out-of-order execution

25

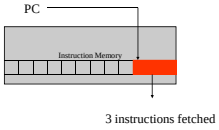
Impediments to High IPC



26

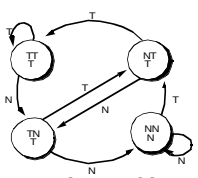
Instruction Flow

- Objective: Fetch multiple instructions per cycle
- Challenges:
  - Branches: control dependences
    - Predict target and direction
  - Branch target misalignment
    - Target near end of line
    - Alignment hardware, etc.
  - Instruction cache misses
    - Cache organization
    - Prefetching, etc.



27

Branch Condition Prediction



Branch inst. address	Information for predict.	Branch target address

- Hardware table remembers
  - History of past several branches encoded by FSM
  - Current state used to generate prediction
- State of the art:
  - Multiple FSMs, dynamically pick “best” one
  - Major topic in 752 and research community

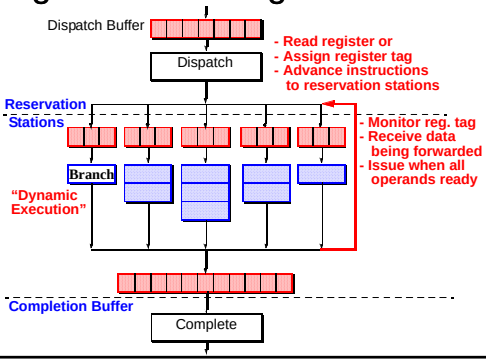
28

Register Data Flow

- Program data dependences cause hazards
  - True dependences (RAW)
  - Antidependences (WAR)
  - Output dependences (WAW)
- When are registers read and written?
  - Out of program order!
  - Hence, any/all of these can occur
- Solution to all three: register renaming

29

Register Renaming



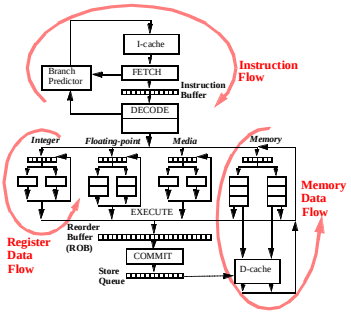
30

Memory Data Flow

- Main impediments:
  - Memory data dependences:
    - WAR/WAW: stores commit in order
      - Hazards not possible. Why?
    - RAW: loads must check pending stores
      - Store queue keeps track of pending store addresses
      - Loads check against these addresses
      - Similar to register bypass logic
      - Comparators are 32 or 64 bits wide (address size)
    - Major source of complexity in modern designs
  - Data cache misses

31

Superscalar Summary



32

Superscalar Summary

- Instruction flow
  - Branches, jumps, calls: predict target, direction
  - Fetch alignment
  - Instruction cache misses
- Register data flow
  - Register renaming: RAW/WAR/WAW
- Memory data flow
  - In-order stores: WAR/WAW
  - Store queue: RAW
  - Data cache misses

33

Memory Hierarchy

- Memory
  - Just an “ocean of bits”
  - Many technologies are available
- Key issues
  - Technology (how bits are stored)
  - Placement (where bits are stored)
  - Identification (finding the right bits)
  - Replacement (finding space for new bits)
  - Write policy (propagating changes to bits)
- Must answer these regardless of memory type

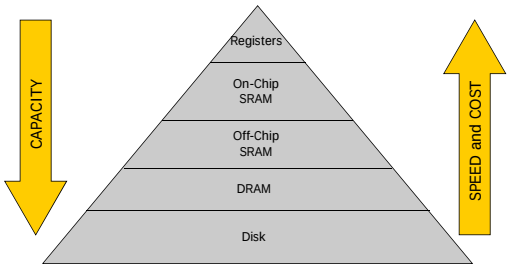
34

Types of Memory

Type	Size	Speed	Cost/bit
Register	< 1KB	< 1ns	$$$$
On-chip SRAM	8KB-6MB	< 10ns	$$$
Off-chip SRAM	1Mb – 16Mb	< 20ns	$$
DRAM	64MB – 1TB	< 100ns	$
Disk	40GB – 1PB	< 20ms	~0

35

Memory Hierarchy



36

Why Memory Hierarchy?

- Need lots of bandwidth

$$BW = \frac{1.0inst}{cycle} \times \left[\frac{1Ifetch}{inst} \times \frac{4B}{Ifetch} + \frac{0.4Dref}{inst} \times \frac{4B}{Dref} \right] \times \frac{1Gcycles}{sec}$$
$$= \frac{5.6GB}{sec}$$

- Need lots of storage
  - 64MB (minimum) to multiple TB
- Must be cheap per bit
  - (TB x anything) is a lot of money!
- These requirements seem incompatible

37

Why Memory Hierarchy?

- Fast and small memories
  - Enable quick access (fast cycle time)
  - Enable lots of bandwidth (1+ L/S/I-fetch/cycle)
- Slower larger memories
  - Capture larger share of memory
  - Still relatively fast
- Slow huge memories
  - Hold rarely-needed state
  - Needed for correctness
- All together: provide appearance of large, fast memory with cost of cheap, slow memory

38

Why Does a Hierarchy Work?

- Locality of reference
  - Temporal locality
    - Reference same memory location repeatedly
  - Spatial locality
    - Reference near neighbors around the same time
- Empirically observed
  - Significant!
  - Even small local storage (8KB) often satisfies >90% of references to multi-MB data set

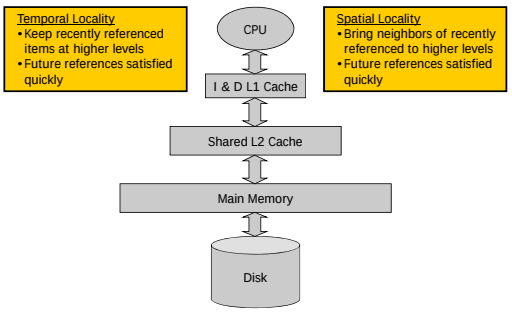
39

Why Locality?

- Analogy:
  - Library (Disk)
  - Bookshelf (Main memory)
  - Stack of books on desk (off-chip cache)
  - Opened book on desk (on-chip cache)
- Likelihood of:
  - Referring to same book or chapter again?
    - Probability decays over time
    - Book moves to bottom of stack, then bookshelf, then library
  - Referring to chapter n+1 if looking at chapter n?

40

Memory Hierarchy



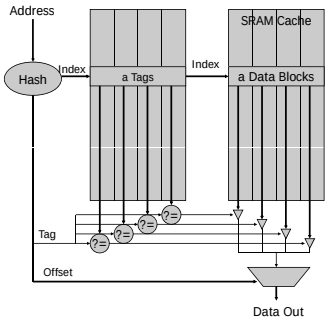
41

Four Burning Questions

- These are:
  - Placement
    - Where can a block of memory go?
  - Identification
    - How do I find a block of memory?
  - Replacement
    - How do I make space for new blocks?
  - Write Policy
    - How do I propagate changes?
- Consider these for caches
  - Usually SRAM
- Will consider main memory, disks later

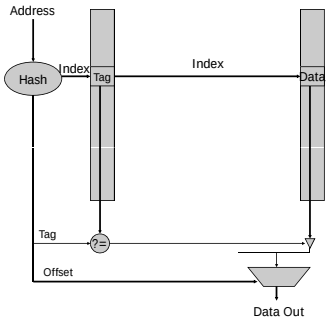
42

Caches: Set-associative



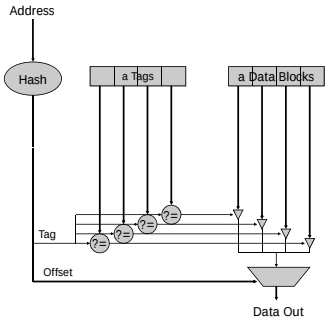
43

Caches: Direct-Mapped



44

Caches: Fully-associative



45

Placement and Identification

32-bit Address

Portion	Length	Purpose
Offset	o=$\log_2$(block size)	Select word within block
Index	i=$\log_2$(number of sets)	Select set of blocks
Tag	t=32 - o - i	ID block within set

- Consider: <BS=block size, S=sets, B=blocks>
  - <64,64,64>: o=6, i=6, t=20: direct-mapped (S=B)
  - <64,16,64>: o=6, i=4, t=22: 4-way S-A (S = B / 4)
  - <64,1,64>: o=6, i=0, t=26: fully associative (S=1)
- Total size = BS x B = BS x S x (B/S)

46

Cache Example

- 32B Cache: <BS=4,S=4,B=8>
  - o=2, i=2, t=2; 2-way set-associative
  - Initially empty
  - Only tag array shown on right
- Trace execution of:

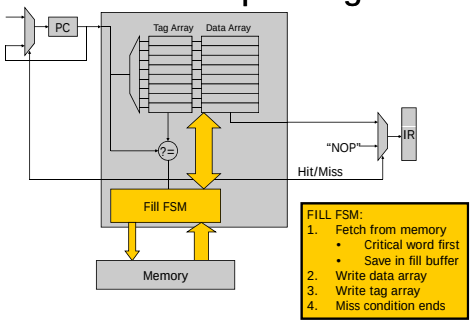
Reference	Binary	Set/Way	Hit/Miss
Load 0x2A	101010	2/0	Miss
Load 0x2B	101011	2/0	Hit
Load 0x3C	111100	3/0	Miss
Load 0x20	100000	0/0	Miss
Load 0x33	110011	0/1	Miss
Load 0x11	010001	0/0 (lru)	Miss/Evict
Store 0x29	101001	2/0	Hit/Dirty

Tag Array

Tag0	Tag1	LRU
01	11	1
		0
10 d		1
11		1

47

I-Caches and Pipelining



48

D-Caches and Pipelining

- Pipelining loads from cache
  - Hit/Miss signal from cache
  - Stalls pipeline or inject NOPs?
    - Hard to do in current real designs, since wires are too slow for global stall signals
  - Instead, treat more like branch misprediction
    - Cancel/flush pipeline
    - Restart when cache fill logic is done

49

D-Caches and Pipelining

- Stores more difficult
  - MEM stage:
    - Perform tag check
    - Only enable write on a hit
    - On a miss, must not write (data corruption)
  - Problem:
    - Must do tag check and data array access sequentially
    - This will hurt cycle time
  - Better solutions exist
    - Beyond scope of this course
    - If you want to do a data cache in your project, come talk to me!

50

Caches and Performance

- Caches
  - Enable design for common case: cache hit
    - Cycle time, pipeline organization
    - Recovery policy
  - Uncommon case: cache miss
    - Fetch from next level
      - Apply recursively if multiple levels
    - What to do in the meantime?
- What is performance impact?
- Various optimizations are possible

51

Cache Misses and Performance

- How does this affect performance?
- Performance = Time / Program

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(code size) (CPI) (cycle time)

- Cache organization affects cycle time
  - Hit latency
- Cache misses affect **CPI**

52

Cache Misses and CPI

$$CPI = \frac{cycles_{hit}}{inst} + \sum_{l=1}^{n} P_l \times MPI_l$$

- P_l is miss penalty at each of n levels of cache
- MPI_l is miss rate per instruction at each of n levels of cache
- Miss rate specification:
  - Per instruction: easy to incorporate in CPI
  - Per reference: must convert to per instruction
    - Local: misses per local reference
    - Global: misses per ifetch or load or store

53

Cache Miss Rate

- Determined by:
  - Program characteristics
    - Temporal locality
    - Spatial locality
  - Cache organization
    - Block size, associativity, number of sets

54

Cache Miss Rates: 3 C's [Hill]

- Compulsory miss
  - First-ever reference to a given block of memory
- Capacity
  - Working set exceeds cache capacity
  - Useful blocks (with future references) displaced
- Conflict
  - Placement restrictions (not fully-associative) cause useful blocks to be displaced
  - Think of as *capacity within set*

55

Caches Summary

- Four questions
  - Placement
    - Direct-mapped, set-associative, fully-associative
  - Identification
    - Tag array used for tag check
  - Replacement
    - LRU, FIFO, Random
  - Write policy
    - Write-through, writeback

56

Caches Summary

$$CPI = \frac{cycles_{hit}}{inst} + \sum_{l=1}^{n} P_l \times MPI_l$$

- Hit latency
  - Block size, associativity, number of blocks
- Miss penalty
  - Overhead, fetch latency, transfer, fill
- Miss rate
  - 3 C's: compulsory, capacity, conflict
  - Determined by locality, cache organization

57

Register File

- Registers managed by programmer/compiler
  - Assign variables, temporaries to registers
  - Limited name space matches available storage
  - Learn more in CS536, CS701

Placement	Flexible (subject to data type)
Identification	Implicit (name == location)
Replacement	Spill code (store to stack frame)
Write policy	Write-back (store on replacement)

58

Main Memory and Virtual Memory

- Use of virtual memory
  - Main memory becomes another level in the memory hierarchy
  - Enables programs with address space or working set that exceed physically available memory
    - No need for programmer to manage overlays, etc.
    - Sparse use of large address space is OK
  - Allows multiple users or programs to timeshare limited amount of physical memory space and address space
- Bottom line: efficient use of expensive resource, and ease of programming

59

Virtual Memory

- Enables
  - Use more memory than system has
  - Think program is only one running
    - Don't have to manage address space usage across programs
    - E.g. think it always starts at address 0x0
  - Memory protection
    - Each program has private VA space: no-one else can clobber it
  - Better performance
    - Start running a large program before all of it has been loaded from disk

60

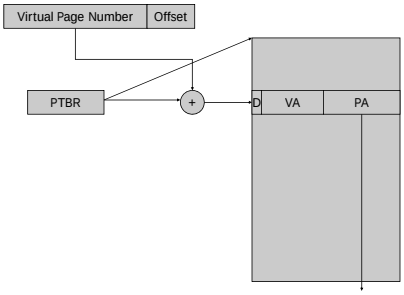
Address Translation

VA	PA	Dirty	Ref	Protection
0x20004000	0x2000	Y/N	Y/N	Read/Write/ Execute

- O/S and hardware communicate via PTE
- How do we find a PTE?
  - &PTE = PTBR + page number * sizeof(PTE)
  - PTBR is private for each program
    - Context switch replaces PTBR contents

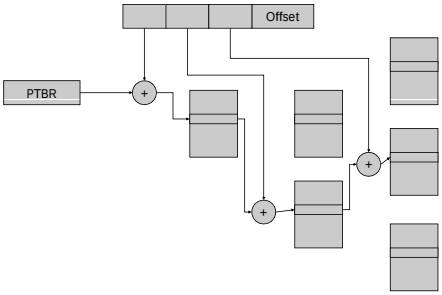
61

Address Translation



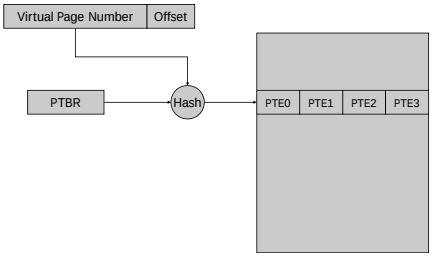
62

Multilevel Page Table



63

Hashed Page Table



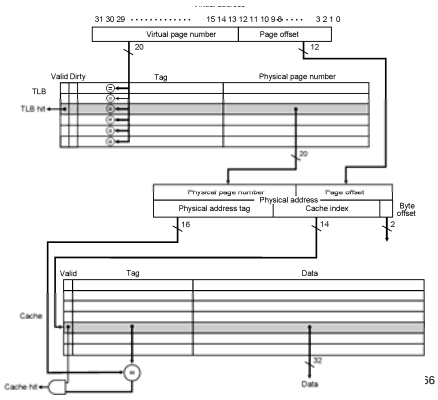
64

High-Performance VM

- VA translation
  - Additional memory reference to PTE
  - Each instruction fetch/load/store now 2 memory references
    - Or more, with multilevel table or has collisions
  - Even if PTE are cached, still slow
- Hence, use special-purpose cache for PTEs
  - Called TLB (translation lookaside buffer)
  - Caches PTE entries
  - Exploits temporal and spatial locality (just a cache)

65

TLB



66

Virtual Memory Protection

- Each process/program has private virtual address space
  - Automatically protected from rogue programs
- Sharing is possible, necessary, desirable
  - Avoid copying, staleness issues, etc.
- Sharing in a controlled manner
  - Grant specific permissions
    - Read
    - Write
    - Execute
    - Any combination
  - Store permissions in PTE and TLB

67

VM Sharing

- Share memory locations by:
  - Map shared physical location into both address spaces:
    - E.g. PA 0xC00DA becomes:
      - VA 0x2D000DA for process 0
      - VA 0x4D000DA for process 1
  - Either process can read/write shared location
- However, causes synonym problem

68

VA Synonyms

- Virtually-addressed caches are desirable
  - No need to translate VA to PA before cache lookup
  - Faster hit time, translate only on misses
- However, VA synonyms cause problems
  - Can end up with two copies of same physical line
- Solutions:
  - Flush caches/TLBs on context switch
  - Extend cache tags to include PID
    - Effectively a shared VA space (PID becomes part of address)

69

Error Detection and Correction

- Main memory stores a huge number of bits
  - Probability of bit flip becomes nontrivial
  - Bit flips (called soft errors) caused by
    - Slight manufacturing defects
    - Gamma rays and alpha particles
    - Interference
    - Etc.
  - Getting worse with smaller feature sizes
- Reliable systems must be protected from soft errors via ECC (error correction codes)
  - Even PCs support ECC these days

70

Error Correcting Codes

- Probabilities:
  - P(1 word no errors) > P(single error) > P(two errors) >> P(>2 errors)
- Detection - signal a problem
- Correction - restore data to correct value
- Most common
  - Parity - single error detection
  - SECDED - single error correction; double bit detection

71

1-bit ECC

Power	Correct	#bits	Comments
Nothing	0,1	1	
SED	00,11	2	01,10 detect errors
SEC	000,111	3	001,010,100 => 0 110,101,011 => 1
SECDED	0000,1111	4	One 1 => 0 Two 1's => error Three 1's => 1

72

ECC

- Reduced overhead by doing codes on word, not bit

# bits	SED overhead	SECDED overhead
1	1 (100%)	3 (300%)
32	1 (3%)	7 (22%)
64	1 (1.6%)	8 (13%)
n	1 (1/n)	1 + $\log_2$ n + a little

73

64-bit ECC

- 64 bits data with 8 check bits
  dddd…..d cccccccc
- Use eight by 9 SIMMS = 72 bits
- Intuition
  - One check bit is parity
  - Other check bits point to
    - Error in data, or
    - Error in all check bits, or
    - No error

74

ECC

- To store (write)
  - Use $data_0$ to compute $check_0$
  - Store $data_0$ and $check_0$
- To load
  - Read $data_1$ and $check_1$
  - Use $data_1$ to compute $check_2$
  - Syndrome = $check_1$ xor $check_2$
    - I.e. make sure check bits are equal

75

4-bit SECDED Example

$C_1 = b_1 \oplus b_2 \oplus b_4$
$C_2 = b_1 \oplus b_3 \oplus b_4$
$C_3 = b_2 \oplus b_3 \oplus b_4$
$P = even\_parity$

Bit Position	1	2	3	4	5	6	7	8	
Codeword	C_1	C_2	b_1	C_3	b_2	b_3	b_4	P	
Original data	1	0	1	1	0	1	0	0	Syndrome
No corruption	1	0	1	1	0	1	0	0	0 0 0, P ok
1 bit corrupted	1	0	0	1	0	1	0	0	0 1 1, P !ok
2 bits corrupted	1	0	0	1	1	1	0	0	1 1 0, P ok

- 4 data bits, 3 check bits, 1 parity bit
- Syndrome is xor of check bits $C_{1\text{-}3}$
  - If (syndrome==0) and (parity OK) => no error
  - If (syndrome != 0) and (parity !OK) => flip bit position pointed to by syndrome
  - If syndrome != 0) and (parity OK) => double-bit error

76

Memory Hierarchy Summary

- Memory hierarchy: Register file
  - Under compiler/programmer control
  - Complex register allocation algorithms to optimize utilization
- Memory hierarchy: Virtual Memory
  - Placement: fully flexible
  - Identification: through page table
  - Replacement: approximate LRU or LFU
  - Write policy: write-through

77

VM Summary

- Page tables
  - Forward page table
    - &PTE = PTBR + VPN * sizeof(PTE)
  - Multilevel page table
    - Tree structure enables more compact storage for sparsely populated address space
  - Inverted or hashed page table
    - Stores PTE for each real page instead of each virtual page
    - HPT size scales up with physical memory
  - Also used for protection, sharing at page level

78

Main Memory Summary

- TLB
  - Special-purpose cache for PTEs
  - Often accessed in parallel with L1 cache
- Main memory design
  - Commodity DRAM chips
  - Wide design space for
    - Minimizing cost, latency
    - Maximizing bandwidth, storage
  - Susceptible to soft errors
    - Protect with ECC (SECDED)
    - ECC also widely used in on-chip memories, busses

79

I/O Device Examples

Device	I or O?	Partner	Data Rate KB/s
Mouse	I	Human	0.01
Display	O	Human	60,000
Modem	I/O	Machine	2-8
LAN	I/O	Machine	500-6000
Tape	Storage	Machine	2000
Disk	Storage	Machine	2000-100,000

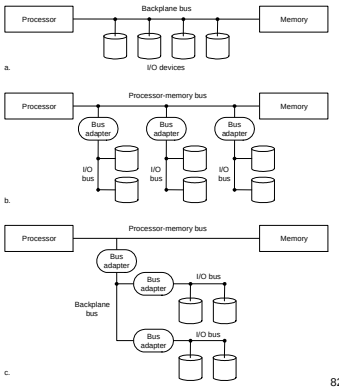
80

I/O Performance

- What is performance?
- Supercomputers read/write 1GB of data
  - Want high bandwidth to vast data (bytes/sec)
- Transaction processing does many independent small I/Os
  - Want high I/O rates (I/Os per sec)
  - May want fast response times
- File systems
  - Want fast response time first
  - Lots of locality

81

Buses in a Computer System



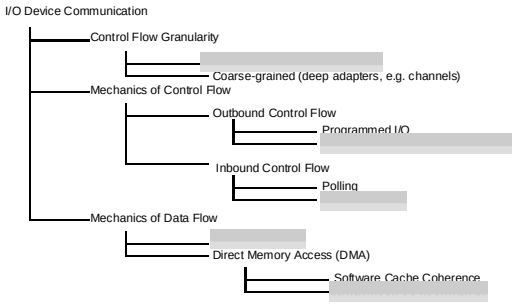
82

Buses

- Synchronous – has clock
  - Everyone watches clock and latches at appropriate phase
  - Transactions take fixed or variable number of clocks
  - Faster but clock limits length
  - E.g. processor-memory
- Asynchronous – requires handshake
  - More flexible
  - I/O

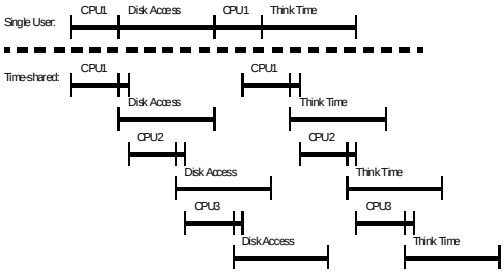
83

Interfacing to I/O Devices



84

Multiprogramming



Summary – I/O

- I/O devices
  - Human interface – keyboard, mouse, display
  - Nonvolatile storage – hard drive, tape
  - Communication – LAN, modem
- Buses
  - Synchronous, asynchronous
  - Custom vs. standard
- Interfacing
  - O/S: protection, virtualization, multiprogramming
  - Interrupts, DMA, cache coherence

Multiprocessor Motivation

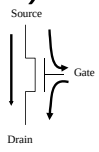
- So far: one processor in a system
- Why not use N processors
  - Higher throughput via parallel jobs
  - Cost-effective
    - Adding 3 CPUs may get 4x throughput at only 2x cost
  - Lower latency from multithreaded applications
    - Software vendor has done the work for you
    - E.g. database, web server
  - Lower latency through parallelized applications
    - Much harder than it sounds

Connect at Memory: Multiprocessors

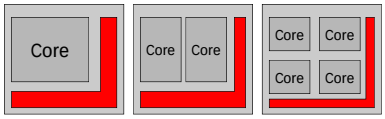
- Shared Memory Multiprocessors
  - All processors can address all physical memory
  - Demands evolutionary operating systems changes
  - Higher throughput with no application changes
  - Low latency, but requires parallelization with proper synchronization
- Most successful: Symmetric MP or SMP
  - 2-64 microprocessors on a bus
  - Too much bus traffic so add caches

Leakage Power (Static/DC)

- Transistors aren't perfect on/off switches
- Even in static CMOS, transistors leak
  - Channel (source/drain) leakage
  - Gate leakage through insulator
    - High-K dielectric replacing SiO_2 helps
- Leakage compounded by
  - Low threshold voltage
    - Low V_{th} => fast switching, more leakage
    - High V_{th} => slow switching, less leakage
  - Higher temperature
    - Temperature increases with power
    - Power increases with C, V^2, A, f
- Rough approximation: leakage proportional to area
  - Transistors aren't free, unless they're turned off
- Controlling leakage
  - Power gating (turn off unused blocks)



Why Multicore



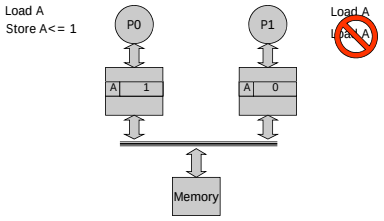
	Single Core	Dual Core	Quad Core
Core area	A	~A/2	~A/4
Core power	W	~W/2	~W/4
Chip power	W + O	W + O'	W + O''
Core performance	P	0.9P	0.8P
Chip performance	P	1.8P	3.2P

Dynamic Power

$$P_{dyn} \approx kCV^2Af$$

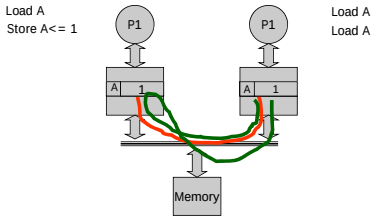
- Aka AC power, switching power
- Static CMOS: current flows when transistors turn on/off
  - Combinational logic evaluates
  - Sequential logic (flip-flop, latch) captures new value (clock edge)
- Terms
  - C: capacitance of circuit (wire length, no. & size of transistors)
  - V: supply voltage
  - A: activity factor
  - f: frequency
- Moore's Law: which terms increase, which decrease?
  - Historically voltage scaling has saved us, but not any more

Cache Coherence Problem



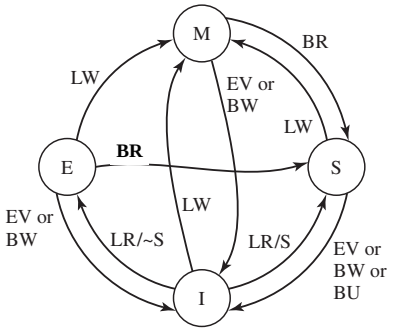
92

Cache Coherence Problem



93

Sample Invalidate Protocol (MESI)

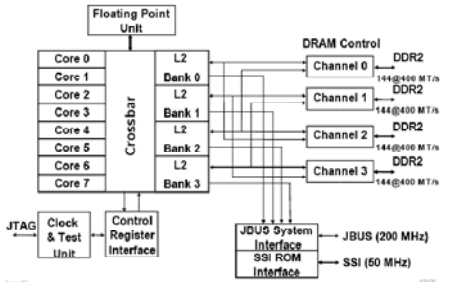


Multithreaded Processors

MT Approach	Resources shared between threads	Context Switch Mechanism
None	Everything	Explicit operating system context switch
Fine-grained	Everything but register file and control logic/state	Switch every cycle
Coarse-grained	Everything but I-fetch buffers, register file and con trol logic/state	Switch on pipeline stall
SMT	Everything but instruction fetch buffers, return address stack, architected register file, control logic/state, reorder buffer, store queue, etc.	All contexts concurrently active; no switching
CMT	Various core components (e.g. FPU), secondary cache, system interconnect	All contexts concurrently active; no switching
CMP	Secondary cache, system interconnect	All contexts concurrently active; no switching

- Many approaches for executing multiple threads on a single die
  - Mix-and-match: IBM Power7 8-core CMP x 4-way SMT

Niagara Block Diagram [Source: J. Laudon]



- 8 in-order cores, 4 threads each
- 4 L2 banks, 4 DDR2 memory controllers

Summary

- Why multicore now?
- Thread-level parallelism
- Shared-memory multiprocessors
  - Coherence
  - Memory ordering
  - Split-transaction buses
- Multithreading
- Multicore processors

Midterm Scope

- Chapter 3.3-3.5:
  - Multiplication, Division, Floating Point
- Chapter 4.10-4.11: Enhancing performance
  - Superscalar lecture notes
  - MIPS R10K reading on course web page
- Chapter 5: Memory Hierarchy
  - Caches, virtual memory
  - SECDED (handout)
- Chapter 6: I/O
- Chapter 5.7-5.9, 7: Multiprocessors
  - Lecture notes on power and multicore
  - Lecture notes on multithreading