

Lec 28: Conclusions

Kavita Bala
CS 3410, Fall 2008
Computer Science
Cornell University

Announcements

- Pizza party was fun
 - Winner: Andrew Cameron and Ross Anderson
- Final project out tomorrow afternoon
 - Demos: Dec 16 (Tuesday)
- Prelim 2: Dec 4 Thursday
 - Hollister 110, 7:30-10:00

Prelim 2 Topics

- Cumulative, but newer stuff:
 - Physical and virtual memory, page tables, TLBs
 - Caches, cache-conscious programming, caching issues
 - Privilege levels, syscalls, traps, interrupts, exceptions
 - Busses, programmed I/O, memory-mapped I/O
 - DMA, disks
 - Synchronization
 - Multicore processors

© Kavita Bala, Computer Science, Cornell University

Goals

- Concurrency poses challenges for:
- Correctness
 - Threads accessing shared memory should not interfere with each other
- Liveness
 - Threads should not get stuck, should make forward progress
- Efficiency
 - Program should make good use of available computing resources (e.g., processors).
- Fairness
 - Resources apportioned fairly between threads

© Kavita Bala, Computer Science, Cornell University

Race conditions

- Def: timing-dependent error involving access to shared state
 - Whether it happens depends on how threads scheduled: who wins “races” to instruction that updates state vs. instruction that accesses state
 - Races are intermittent, may occur rarely
 - Timing dependent = small changes can hide bug
 - A program is correct *only* if *all possible* schedules are safe
 - Number of possible schedule permutations is huge
 - Need to imagine an adversary who switches contexts at the worst possible time

© Kavita Bala, Computer Science, Cornell University

Mutexes

- Critical sections typically associated with mutual exclusion locks (*mutexes*)
- Only one thread can hold a given mutex at a time
- Acquire (lock) mutex on entry to critical section
 - Or block if another thread already holds it
- Release (unlock) mutex on exit
 - Allow one waiting thread (if any) to acquire & proceed

```
pthread_mutex_init(m);  
pthread_mutex_lock(m);    pthread_mutex_lock(m);  
hits = hits+1;           hits = hits+1;  
pthread_mutex_unlock(m); pthread_mutex_unlock(m);
```



© Kavita Bala, Computer Science, Cornell University

Using atomic hardware primitives

- Mutex implementations usually rely on special hardware instructions that *atomically* do a read and a write.
- Requires special memory system support on multiprocessors

Mutex init: lock = false;

```
while (test_and_set(&lock));
```

Critical Section

```
lock = false;
```

test_and_set uses a special hardware instruction that sets the lock and returns the OLD value (true: locked; false: unlocked)
- Alternative instruction: compare & swap, load linked/store conditional
-

Computer Science, Cornell University

Using test-and-set for mutual exclusion

```
boolean lock = false;
```

```
while test_and_set(&lock) skip
```

```
//spin until lock is acquired.
```

```
... do critical section ...
```

```
//only one process can be in this section at a time
```

```
lock = false ;
```

```
// release lock when finished with the
```

```
// critical section
```

```
boolean test_and_set (boolean *lock) {  
    boolean old = *lock;  
    *lock = true;  
    return old;  
}
```

© Kavita Bala, Computer Science, Cornell University

Beyond mutexes

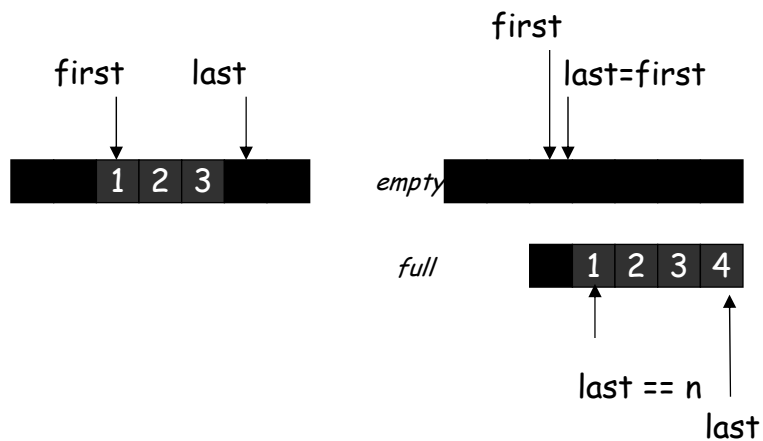
- Sometimes need to share resources in non-exclusive way
- Example: shared queue (multiple readers, multiple writers)
- How to let a reader wait for data without blocking a mutex?

```
char get() {  
    char c = buffer[first];  
    first++;  
}
```

© Kavita Bala, Computer Science, Cornell University

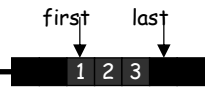
Example: buffer

- Invariant: active cells start at first, end at n



© Kavita Bala, Computer Science, Cornell University

A first broken cut



```
// invariant: data is in buffer[first..last-1].
mutex_t *m = ...;
char buffer[n];
int first = 0, last = 0;

void put(char c) {
    lock(m);
    buffer[last] = c;
    last = (last+1);
    unlock(m);
}

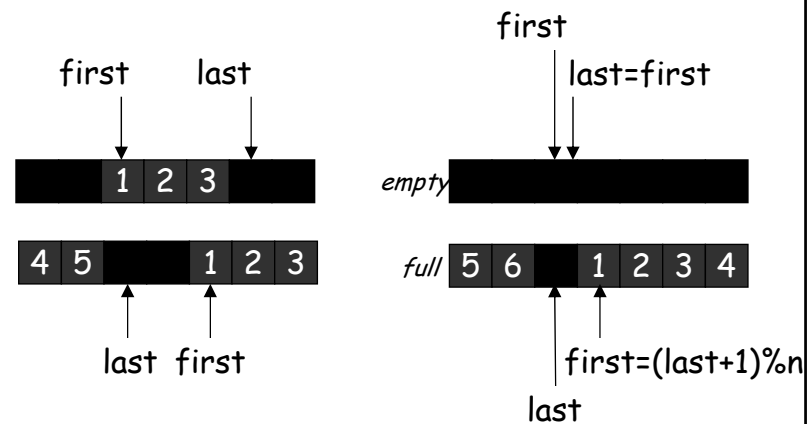
char get() {
    bool done = false;
    while (!done) {
        lock(m);
        if (first == last) {
            unlock(m);
            continue;
        }
        char c = buffer[first];
        first = (first+1);
        unlock(m);
        done = true;
    }
}

// ... Blocks all writers if empty
// ... Reader still spins on empty queue
```

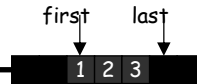
Same issues here for full queue

Example: ring buffer

- A useful data structure for IPC
- Invariant: active cells start at first, end at last-1, last never incremented up to first



A first broken cut



```
// invariant: data is in buffer[first..last-1].
mutex_t *m;
char buffer[n];
int first = 0, last = 0;

char get() {
    bool done = false;
    while (!done) {
        lock(m);
        char c = buffer[first];
        printf("first: %d\n", first);
        first = (first + 1) % n;
        unlock(m);
        done = true;
    }
}

void put(char c) {
    lock(m);
    buffer[last] = c;
    last = (last + 1) % n;
    unlock(m);
}

// Same issues here for full queue
```

Oops! Blocks all writers if empty

Oops! Reader still spins on empty queue

© Kavita Bala, Computer Science, Cornell University

Condition variables

- To let thread wait (not holding the mutex!) until a condition is true, use a *condition variable* [Hoare]
- `wait(m, c)` : atomically release `m` and go to sleep waiting for condition `c`, wake up holding `m`
 - Must be atomic to avoid *wake-up-waiting race*
- `signal(c)` : wake up one thread waiting on `c`
- `broadcast(c)` : wake up all threads waiting on `c`
- POSIX (e.g., Linux): `pthread_cond_wait`, `pthread_cond_signal`, `pthread_cond_broadcast`

© Kavita Bala, Computer Science, Cornell University

Using a condition variable

- `wait(m, c)` : release m, sleep waiting for c, wake up holding m
- `signal(c)` : wake up one thread waiting on c

```
char put(char c) {
    lock(m);
    while ((first-last)%n == 1)
        wait(m, not_full);
    buffer[last] = c;
    last = (last+1)%n;
    unlock(m);
    signal(not_empty);
}

mutex_t *m;
cond_t *not_empty, *not_full;
char get() {
    lock(m);
    while (first == last)
        wait(m, not_empty);
    char c = buffer[first];
    first = (first+1)%n;
    unlock(m);
    signal(not_full);
}
```

© Kavita Bala, Computer Science, Cornell University

Monitors

- A *monitor* is a shared concurrency-safe data structure
- Has one mutex
- Has some number of condition variables
- Operations acquire mutex so only one thread can be in the monitor at a time

- Our buffer implementation is a monitor
- Some languages (e.g. Java, C#) provide explicit support for monitors

© Kavita Bala, Computer Science, Cornell University

Java concurrency

- Java object is a simple monitor
 - Acts as a mutex via `synchronized { S }` statement and `synchronized` methods
 - Has one (!) builtin condition variable tied to the mutex
 - `o.wait()` = `wait(o, o)`
 - `o.notify()` = `signal(o)`
 - `o.notifyAll()` = `broadcast(o)`
 - `synchronized(o) {S}` = `lock(o); S; unlock(o)`
 - Java `wait()` can be called even when mutex is not held. Mutex not held when awoken by `signal()`.
Useful?

© Kavita Bala, Computer Science, Cornell University

More synchronization mechanisms

Implementable with mutexes and condition variables:

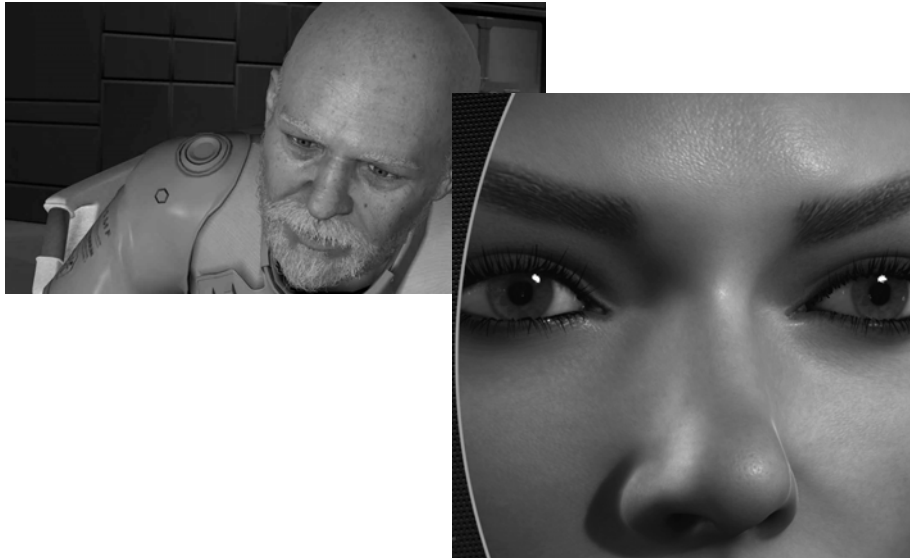
- Reader/writer locks
 - Any number of threads can hold a read lock
 - Only one thread can hold the writer lock
- Semaphores
 - Some number n of threads are allowed to hold the lock
 - $n=1 \Rightarrow$ semaphore = mutex
- Message-passing, sockets
 - `send()/recv()` transfer data and synchronize

© Kavita Bala, Computer Science, Cornell University

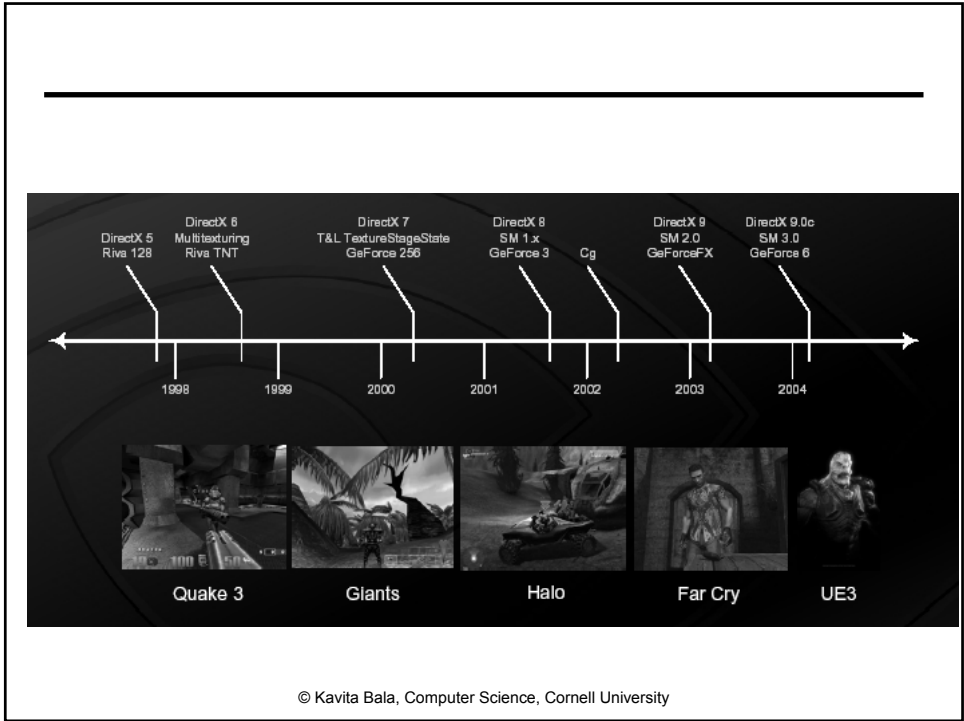
Where are we going?

© Kavita Bala, Computer Science, Cornell University

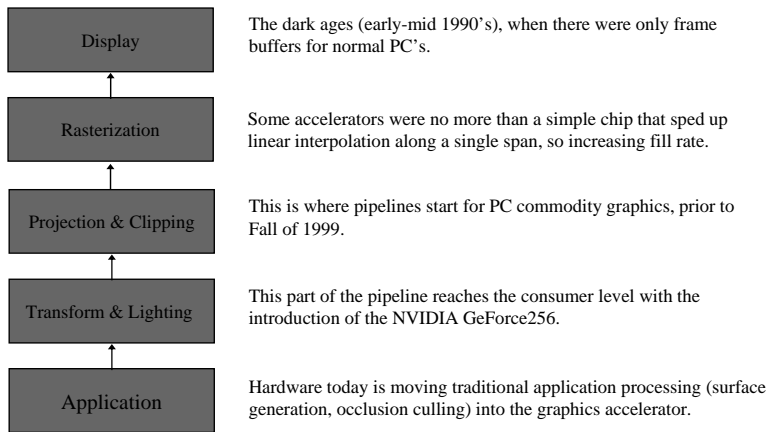
Real-Time Hardware



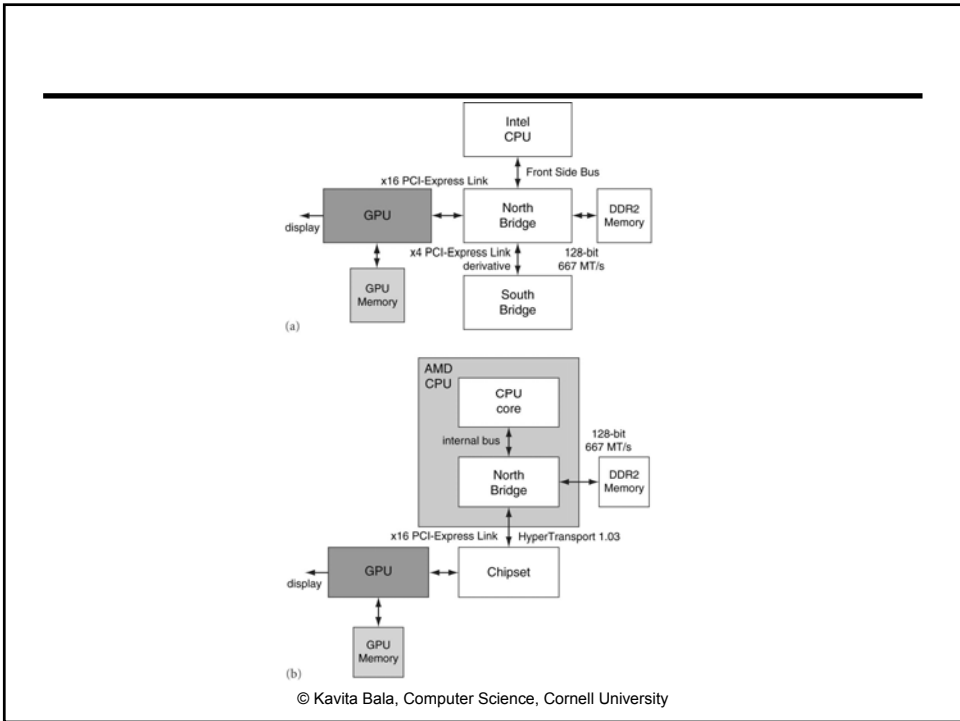
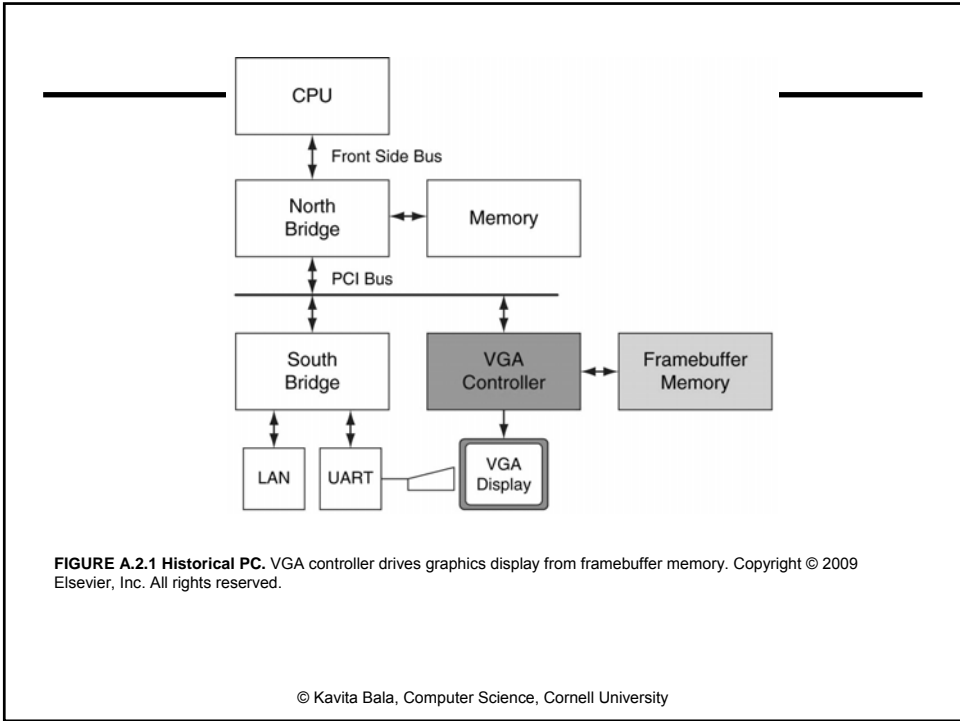
© Kavita Bala, Computer Science, Cornell University



Brief History



© Kavita Bala, Computer Science, Cornell University

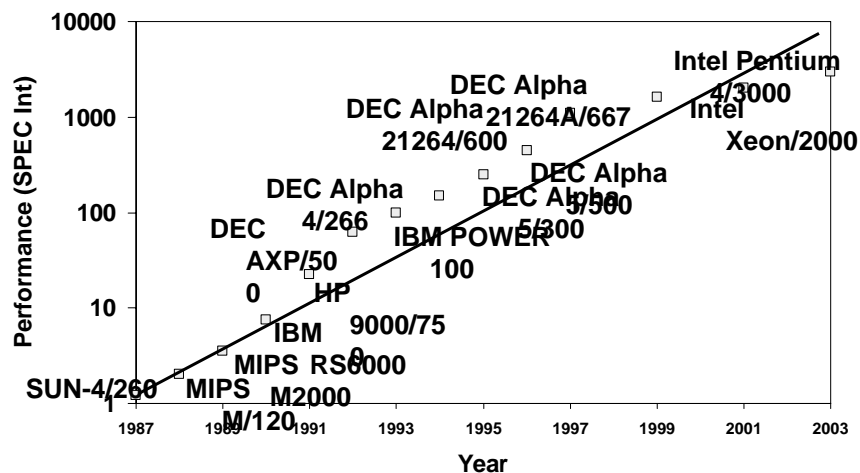


Moore's Law

- 1965
 - number of transistors that can be integrated on a die would double every 18 to 24 months (i.e., grow exponentially with time).
- Amazingly visionary
 - 2300 transistors, 1 MHz clock (Intel 4004) - 1971
 - 16 Million transistors (Ultra Sparc III)
 - 42 Million transistors, 2 GHz clock (Intel Xeon) – 2001
 - 55 Million transistors, 3 GHz, 130nm technology, 250mm² die (Intel Pentium 4) – 2004
 - 290+ Million transistors, 3 GHz (Intel Core 2 Duo) – 2007

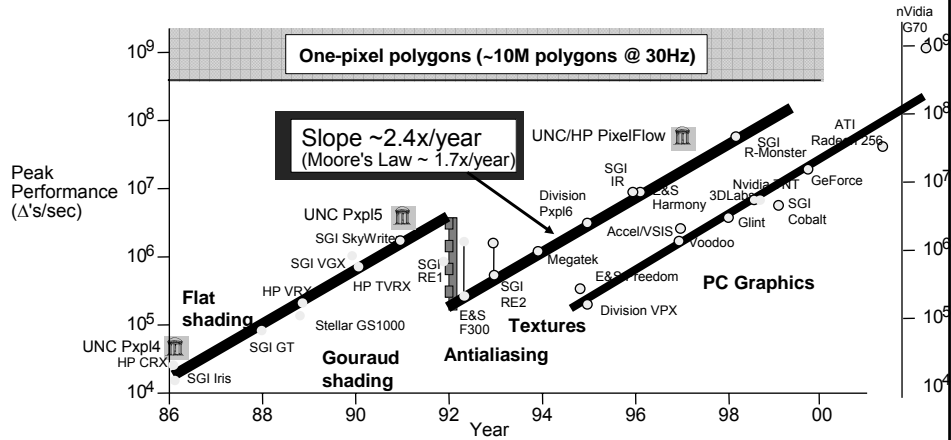
© Kavita Bala, Computer Science, Cornell University

Processor Performance Increase



© Kavita Bala, Computer Science, Cornell University

Faster than Moore's Law



Graph courtesy of Professor John Poulton (from Eric Haines)

© Kavita Bala, Computer Science, Cornell University

NVidia Tesla Architecture



© Kavita Bala, Computer Science, Cornell University

Why are GPUs so fast?

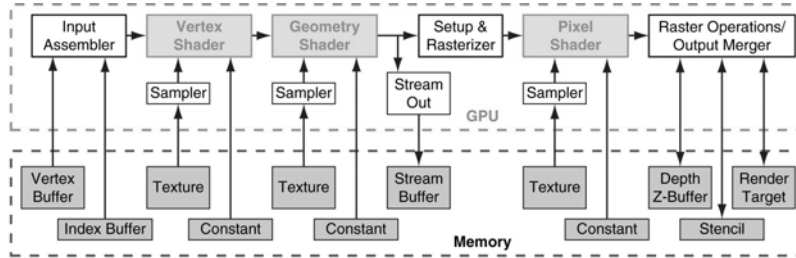


FIGURE A.3.1 Direct3D 10 graphics pipeline. Each logical pipeline stage maps to GPU hardware or to a GPU processor. Programmable shader stages are blue, fixed-function blocks are white, and memory objects are grey. Each stage processes a vertex, geometric primitive, or pixel in a streaming dataflow fashion. Copyright © 2009 Elsevier, Inc. All rights reserved.

- Pipelined and parallel
- Very, very parallel: 128 to 1000 cores

© Kavita Bala, Computer Science, Cornell University

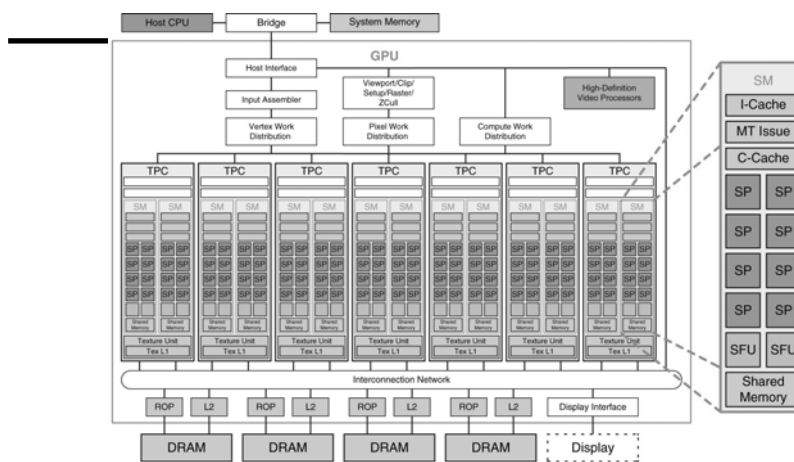


FIGURE A.2.5 Basic unified GPU architecture. Example GPU with 112 streaming processor (SP) cores organized in 14 streaming multiprocessors (SMs); the cores are highly multithreaded. It has the basic Tesla architecture of an NVIDIA GeForce 8800. The processors connect with four 64-bit-wide DRAM partitions via an interconnection network. Each SM has eight SP cores, two special function units (SFUs), instruction and constant caches, a multithreaded instruction unit, and a shared memory. Copyright © 2009 Elsevier, Inc. All rights reserved.

© Kavita Bala, Computer Science, Cornell University

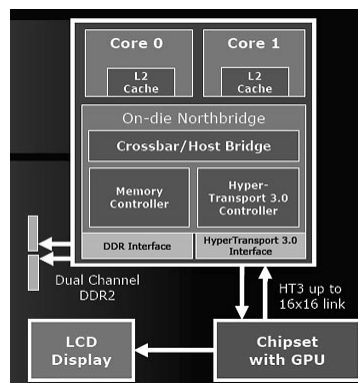
General computing with GPUs

- Can we use these machines for general computation?
- Scientific Computing
 - MATLAB codes
- Convex hulls
- Molecular Dynamics
- Etc.

- CUDA: using it as a general purpose multicore processor

© Kavita Bala, Computer Science, Cornell University

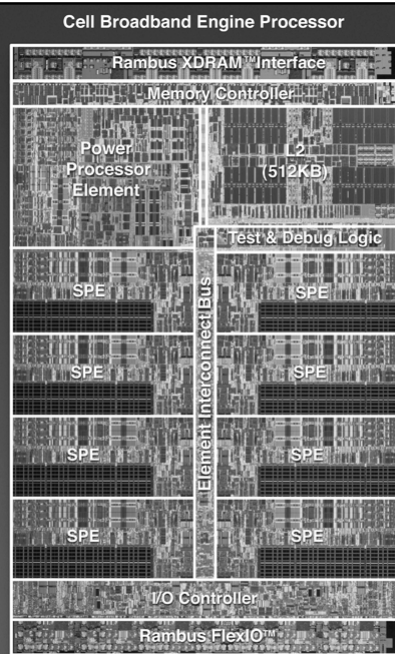
AMDs Hybrid CPU/GPU



© Kavita Bala, Computer Science, Cornell University

Cell

- IBM/Sony/Toshiba
- Sony Playstation 3
- PPE
- SPEs (synergistic)



© Kavita Bala, Computer Science, Cornell University

Classification of Parallelism

- Flynn's taxonomy

		Data Streams	
		Single	Multiple
Instruction Streams	Single	SISD: Intel Pentium 4	SIMD: SSE instructions of x86 Early GPUs
	Multiple	MISD: No examples today	MIMD: Intel Xeon e5345 Cell Tesla

© Kavita Bala, Computer Science, Cornell University

Parallelism

- *Must* exploit parallelism for performance
 - Lot of parallelism in graphics applications
- SIMD: single instruction, multiple data
 - Perform same operation in parallel on many data items
 - Data parallelism
- MIMD: multiple instruction, multiple data
 - Run separate programs in parallel (on different data)
 - Task parallelism

© Kavita Bala, Computer Science, Cornell University

Do you believe?



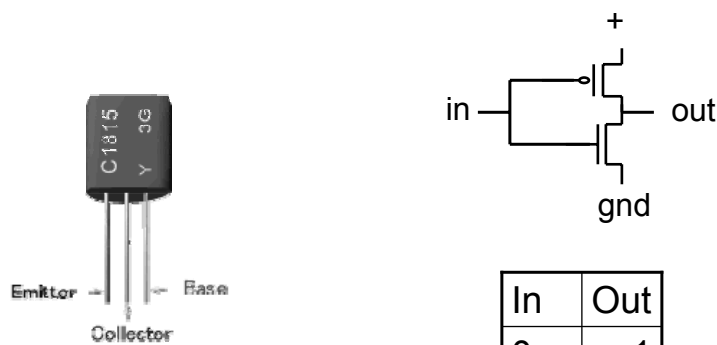
© Kavita Bala, Computer Science, Cornell University

Course Objective

- Bridge the gap between hardware and software
 - How a processor works
 - How a computer is organized
- Establish a foundation for building higher-level applications
 - How to understand program performance
 - How to understand where the world is going

© Kavita Bala, Computer Science, Cornell University

Transistors and Gates



In	Out
0	1
1	0

Truth table

© Kavita Bala, Computer Science, Cornell University

NAND Gate

- Function: NAND
- Symbol:

A	B	out
0	0	1
1	0	1
0	1	1
1	1	0

© Kavita Bala, Computer Science, Cornell University

Building Functions

- NOT:
- AND:
- OR:

- NAND and NOR are universal
 - Can implement any function with NAND or just NOR gates
 - useful for manufacturing

© Kavita Bala, Computer Science, Cornell University

Logic Manipulation

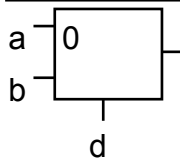
- Can specify functions by describing gates, truth tables or logic equations
- Can manipulate logic equations algebraically
- Can also use a truth table to prove equivalence
- Example: $(a+b)(a+c) = a + bc$

$$\begin{aligned}
 &(a+b)(a+c) \\
 &= aa + ab + ac + bc \\
 &= a + a(b+c) + bc \\
 &= a(1 + (b+c)) + bc \\
 &= a + bc
 \end{aligned}$$

a	b	c	a+b	a+c	LHS	bc	RHS
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	1	1	1	1
1	0	0	1	1	1	0	1
1	0	1	1	1	1	0	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

© Kavita Bala, Computer Science, Cornell University

Multiplexer Implementation

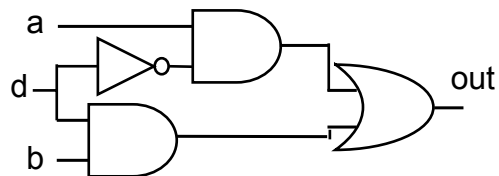


a	b	d	out
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

- Draw the circuit

d \ ab	00	01	11	10
0	0	0	1	1
1	0	1	1	0

$$- \text{out} = a\bar{d} + bd$$



© Kavita Bala, Computer Science, Cornell University

Binary Representation

- $37 = 32 + 4 + 1$

$$\begin{array}{ccccccc} 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ \hline 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\ 64 & 32 & 16 & 8 & 4 & 2 & 1 \end{array}$$

© Kavita Bala, Computer Science, Cornell University

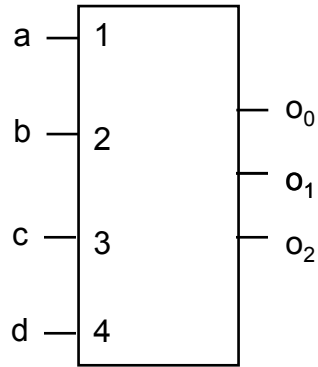
Hexadecimal Representation

$$\begin{array}{cc} 25 \\ \hline 16^1 & 16^0 \end{array}$$

- 37 decimal = $(25)_{16}$
- Convention
 - Base 16 is written with a leading 0x
 - $37 = 0x25$
- Need extra digits!
 - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- Binary to hexadecimal is easy
 - Divide into groups of 4, translate groupwise into hex digits

© Kavita Bala, Computer Science, Cornell University

Encoder Truth Table



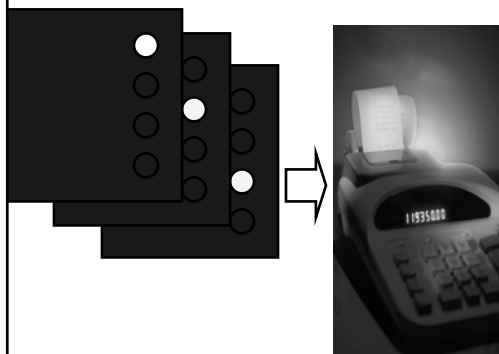
A 3-bit
encoder
with 4 inputs
for simplicity.

a	b	c	d		o2	o1	o0
0	0	0	0		0	0	0
1	0	0	0		0	0	1
0	1	0	0		0	1	0
0	0	1	0		0	1	1
0	0	0	1		1	0	0

- $o2 = \overline{a}b\overline{c}d$
- $o1 = \overline{a}bcd + a\overline{b}c\overline{d}$
- $o0 = abcd + a\overline{b}cd$

© Kavita Bala, Computer Science, Cornell University

Ballot Reading



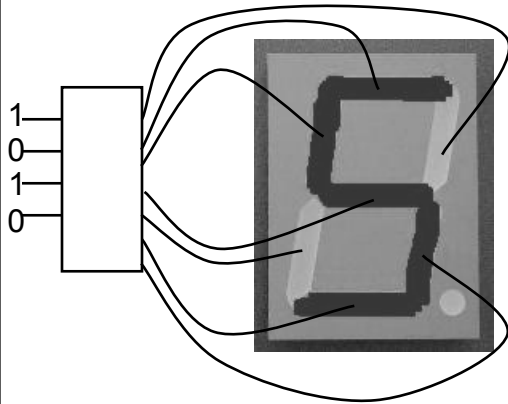
Ballots

The 3410 voting
machine

- Ok, we built first half of the machine
- Need to display the result

© Kavita Bala, Computer Science, Cornell University

7-Segment LED Decoder

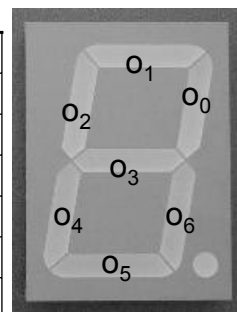


- 4 inputs encoded in binary
- 8 outputs, each driving an independent, rectangular LED
- Can display numbers

© Kavita Bala, Computer Science, Cornell University

7-Segment Decoder Truth Table

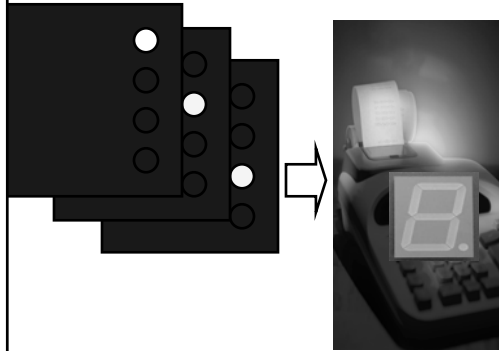
i_3	i_2	i_1	i_0	o_0	o_1	o_2	o_3	o_4	o_5	o_6
0	0	0	0	1	1	1	0	1	1	1
0	0	0	1	1	0	0	0	0	0	1
0	0	1	0	1	1	0	1	1	1	0
0	0	1	1	1	1	0	1	0	1	1
0	1	0	0	1	0	1	1	0	0	1
0	1	0	1	0	1	1	1	0	1	1
0	1	1	0	0	0	1	1	1	1	1
0	1	1	1	1	1	0	0	0	0	1
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1



© Kavita Bala, Computer Science, Cornell University

Ballot Reading

- Done!



Ballots

The 3410 voting machine

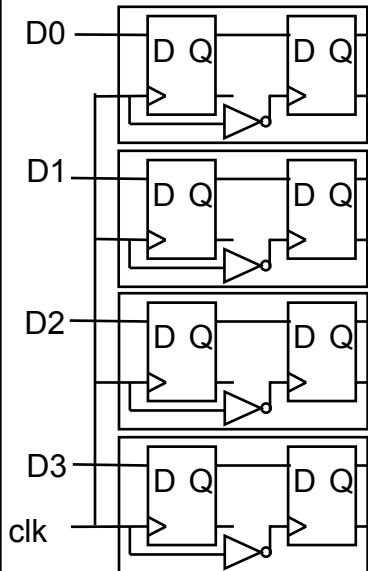
© Kavita Bala, Computer Science, Cornell University

Stateful Components

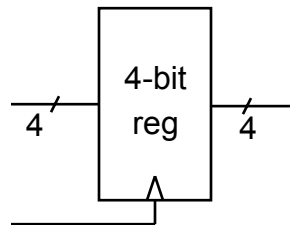
- Until now is combinatorial logic
 - Output is computed when inputs are present
 - System has no internal state
 - Nothing computed in the present can depend on what happened in the past!
- Need a way to record data
- Need a way to build stateful circuits
- Need a state-holding device

© Kavita Bala, Computer Science, Cornell University

Registers



- A register is simply a set of master-slave flip-flops in parallel with a shared clock



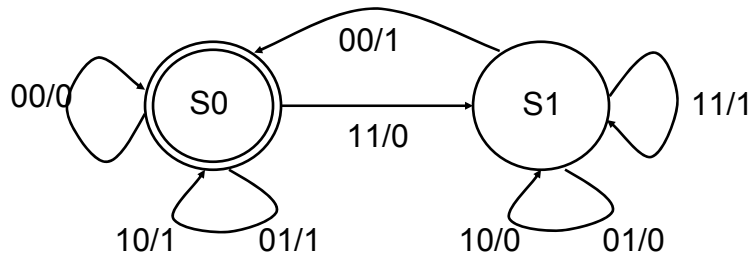
© Kavita Bala, Computer Science, Cornell University

Summary

- We can now build interesting devices with sensors
 - Using combinatorial logic
- We can also store data values
 - In state-holding elements
 - Coupled with clocks

© Kavita Bala, Computer Science, Cornell University

FSM: State Diagram

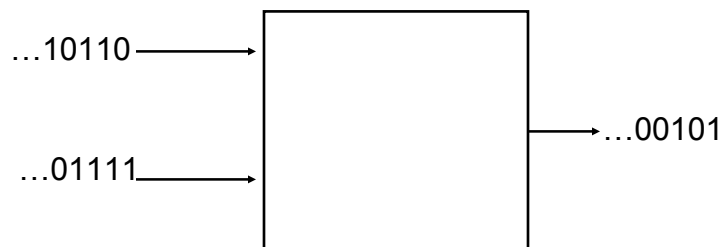


- Two states: S0 (no carry), S1 (carry in hand)
- Inputs: a and b
- Output: z
 - Arcs labelled with input bits a and b, and output z

© Kavita Bala, Computer Science, Cornell University

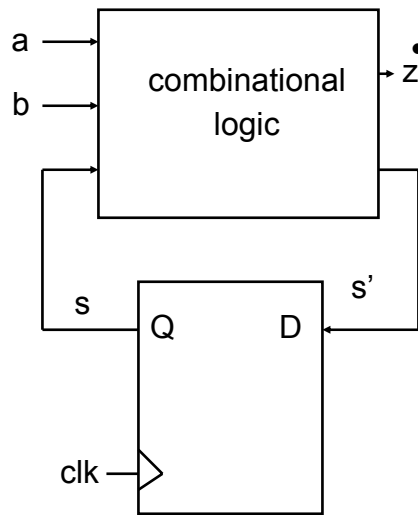
FSM: Serial Adder

- Add two input bit streams
 - streams are sent with least-significant-bit (lsb) first



© Kavita Bala, Computer Science, Cornell University

Serial Adder: Circuit



• Equations

$$- z = \bar{a}\bar{b}s + \bar{a}b\bar{s} + a\bar{b}\bar{s} + abs$$

$$- s' = \bar{a}bs + \bar{a}b\bar{s} + a\bar{b}\bar{s} + abs$$

© Kavita Bala, Computer Science, Cornell University

Binary Arithmetic

$$\begin{array}{r} 12 \\ + 25 \\ \hline 37 \end{array}$$

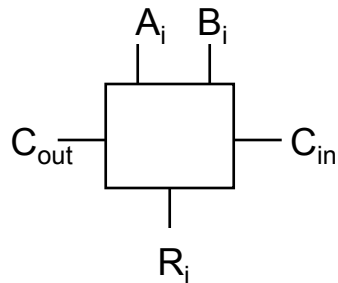
- Arithmetic works the same way regardless of base
 - Add the digits in each position
 - Propagate the carry

$$\begin{array}{r} 001100 \\ + 011010 \\ \hline 100110 \end{array}$$

- Unsigned binary addition is pretty easy
 - Combine two bits at a time
 - Along with a carry

© Kavita Bala, Computer Science, Cornell University

1-bit Adder with Carry

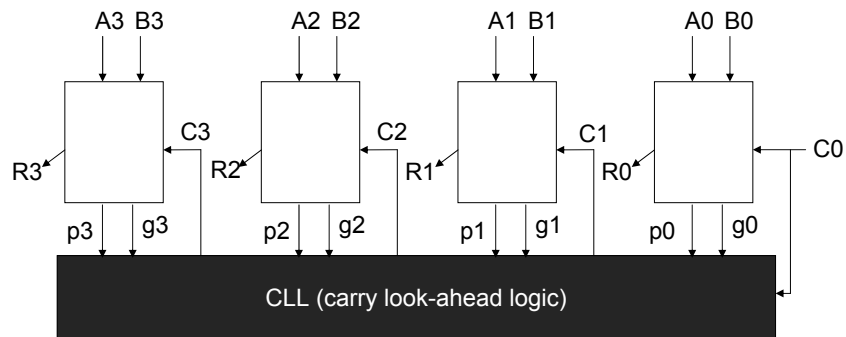


C_i	A_i	B_i	C_{out}	R_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

- Adds two 1-bit numbers, along with carry-in, computes 1-bit result and carry out
- Can be cascaded to add N-bit numbers

© Kavita Bala, Computer Science, Cornell University

4-bit CLA



- Given A,B's, all p,g's are generated in 1 gate delay in parallel.
- Given all p,g's, all C's are generated in 2 gate delay in parallel.
- Given all C's, all R's are generated in 2 gate delay in parallel.
- Sequential operation in RCA is made into parallel operation!!

© Kavita Bala, Computer Science, Cornell University

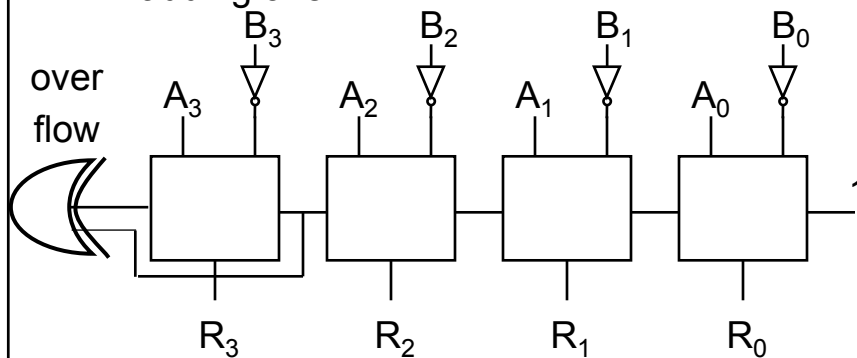
Two's Complement

- Nonnegative numbers are represented as usual
 - 0 = 0000
 - 1 = 0001
 - 3 = 0011
 - 7 = 0111
- To negate a number, flip all bits, add one
 - -1: 1 \Rightarrow 0001 \Rightarrow 1110 \Rightarrow 1111
 - -3: 3 \Rightarrow 0011 \Rightarrow 1100 \Rightarrow 1101
 - -7: 7 \Rightarrow 0111 \Rightarrow 1000 \Rightarrow 1001
 - -8: 8 \Rightarrow 1000 \Rightarrow 0111 \Rightarrow 1000
 - -0: 0 \Rightarrow 0000 \Rightarrow 1111 \Rightarrow 0000 (this is good, -0 = +0)

© Kavita Bala, Computer Science, Cornell University

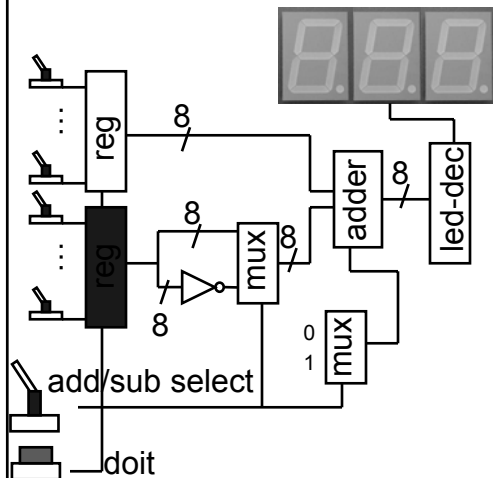
Two's Complement Subtraction

- Subtraction is simply addition, where one of the operands has been negated
 - Negation is done by inverting all bits and adding one



© Kavita Bala, Computer Science, Cornell University

A Calculator

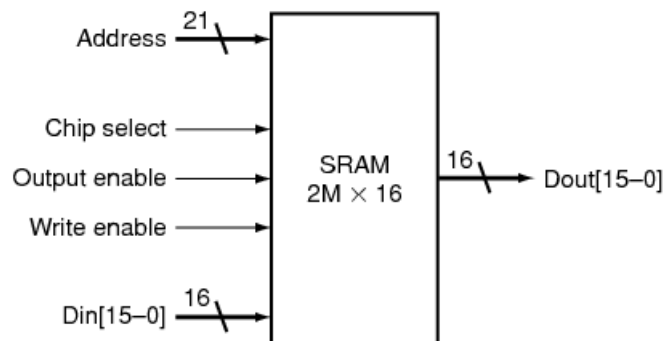


- Enter numbers to be added or subtracted using toggle switches
- Select: ADD or SUBTRACT
- Muxes feed A and B, or A and $-B$, to the 8-bit adder
- The 8-bit decoder for the hex display is straightforward

© Kavita Bala, Computer Science, Cornell University

Static RAM: SRAM

- Static-RAM
 - So called because once stored, data values are stable as long as electricity is supplied
 - Based on regular flip-flops with gates



Parallel Memory Banks

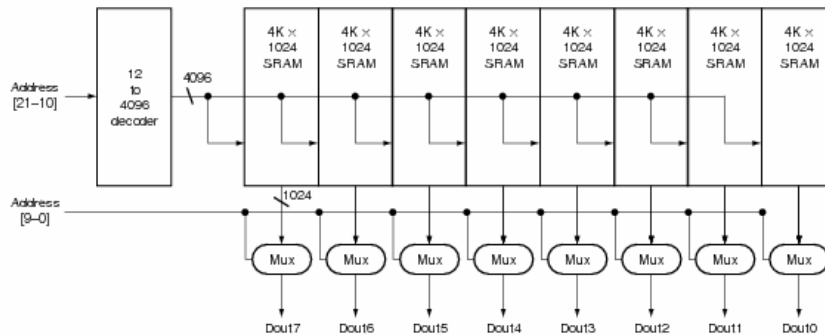


FIGURE B.9.4 Typical organization of a $4M \times 8$ SRAM as an array of $4K \times 1024$ arrays. The first decoder generates the addresses for eight $4K \times 1024$ arrays; then a set of multiplexers is used to select 1 bit from each 1024-bit-wide array. This is a much easier design than a single-level decoder that would need either an enormous decoder or a gigantic multiplexer. In practice, a modern SRAM of this size would probably use an even larger number of blocks, each somewhat smaller.

© Kavita Bala, Computer Science, Cornell University

Dynamic RAM: DRAM

- Dynamic-RAM
 - Data values require constant refresh
 - Internal circuitry keeps capacitor charges

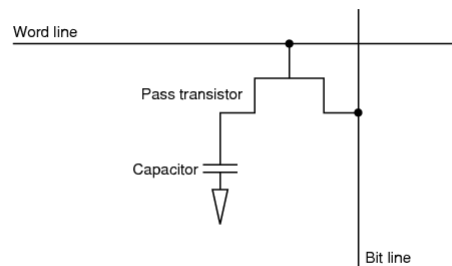
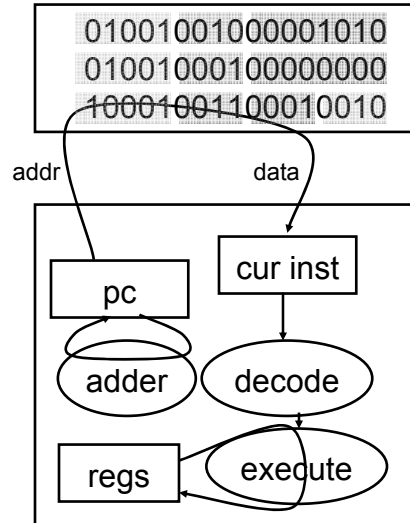


FIGURE B.9.5 A single-transistor DRAM cell contains a capacitor that stores the cell contents and a transistor used to access the cell.

© Kavita Bala, Computer Science, Cornell University

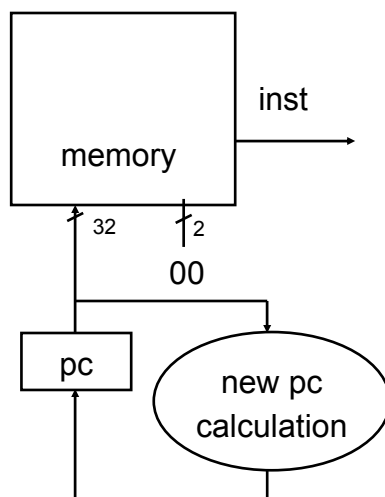
Instruction Usage

- Instructions are stored in memory, encoded in binary
- A basic processor
 - fetches
 - decodes
 - executesone instruction at a time



© Kavita Bala, Computer Science, Cornell University

Instruction Fetch



- Read instruction from memory
- Calculate address of next instruction
- Fetch next instruction

© Kavita Bala, Computer Science, Cornell University

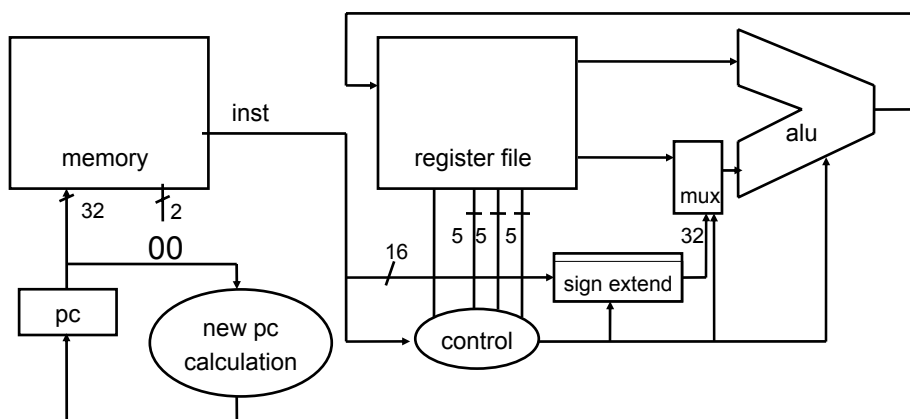
Arithmetic Instructions

op	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- if $op == 0$ && $func == 0x21$
 - $R[rd] = R[rs] + R[rt]$ (unsigned)
- if $op == 0$ && $func == 0x23$
 - $R[rd] = R[rs] - R[rt]$ (unsigned)
- if $op == 0$ && $func == 0x25$
 - $R[rd] = R[rs] | R[rt]$

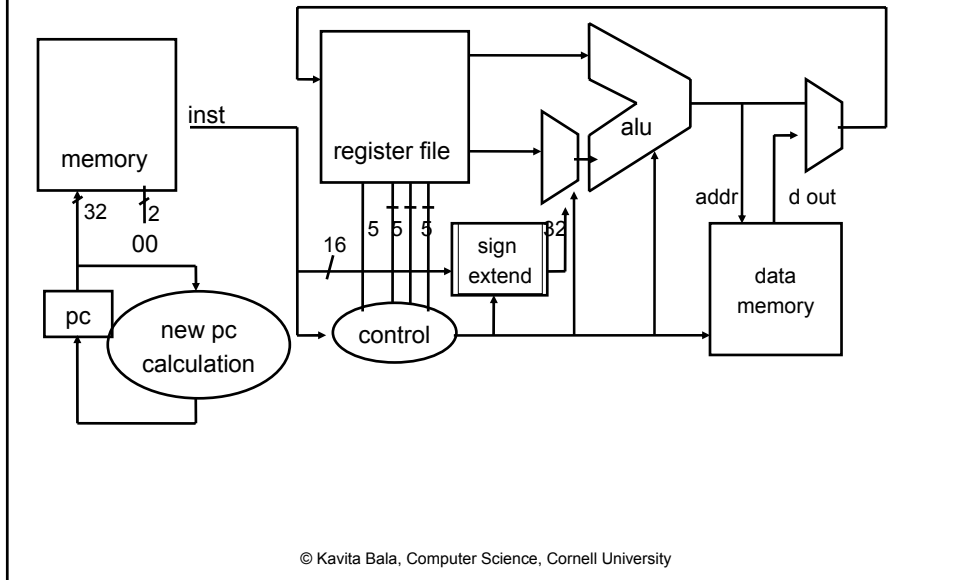
© Kavita Bala, Computer Science, Cornell University

Arithmetic Ops with Immediates

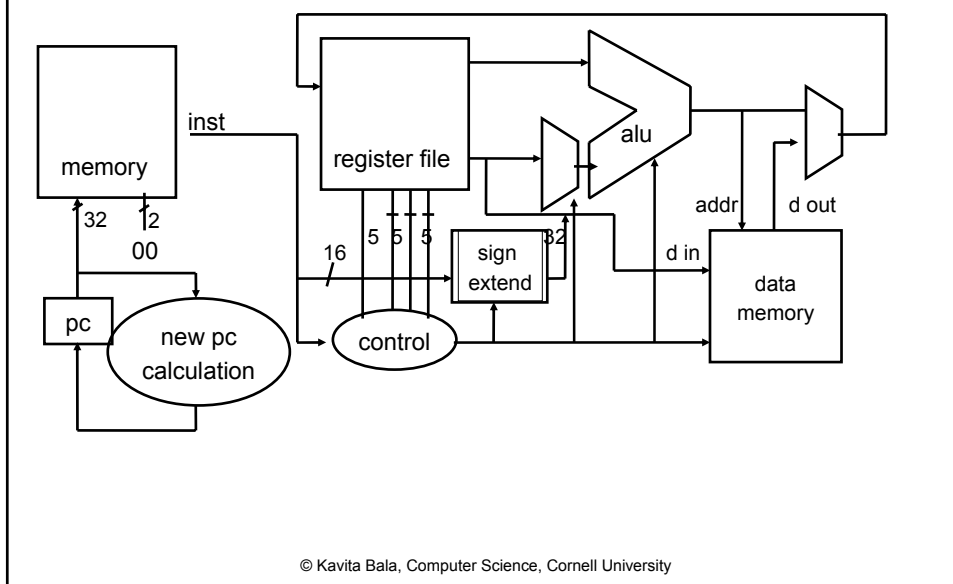


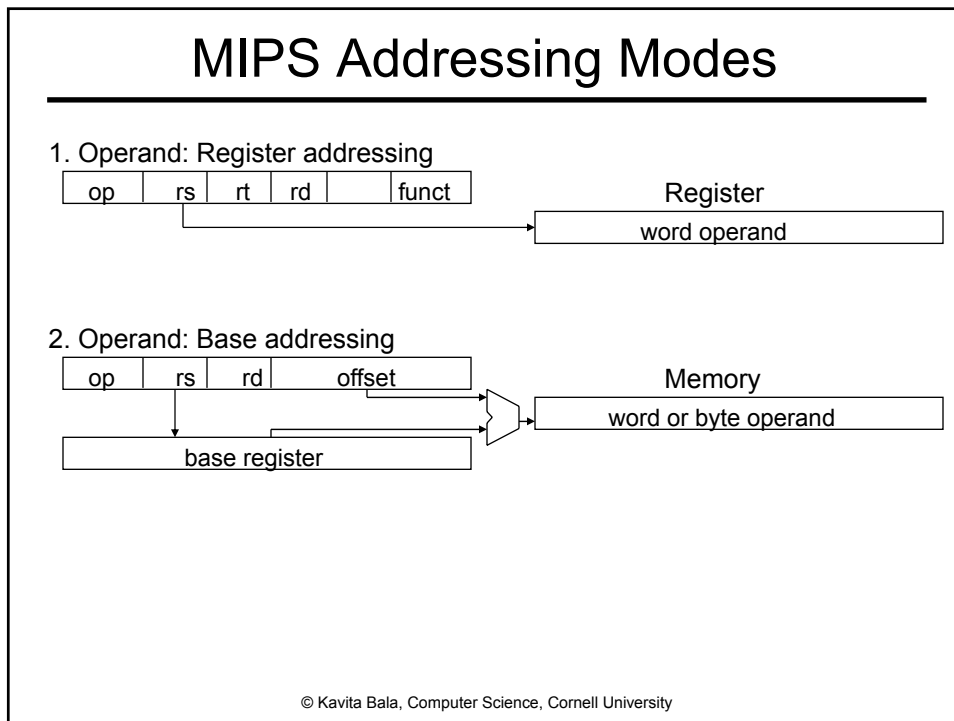
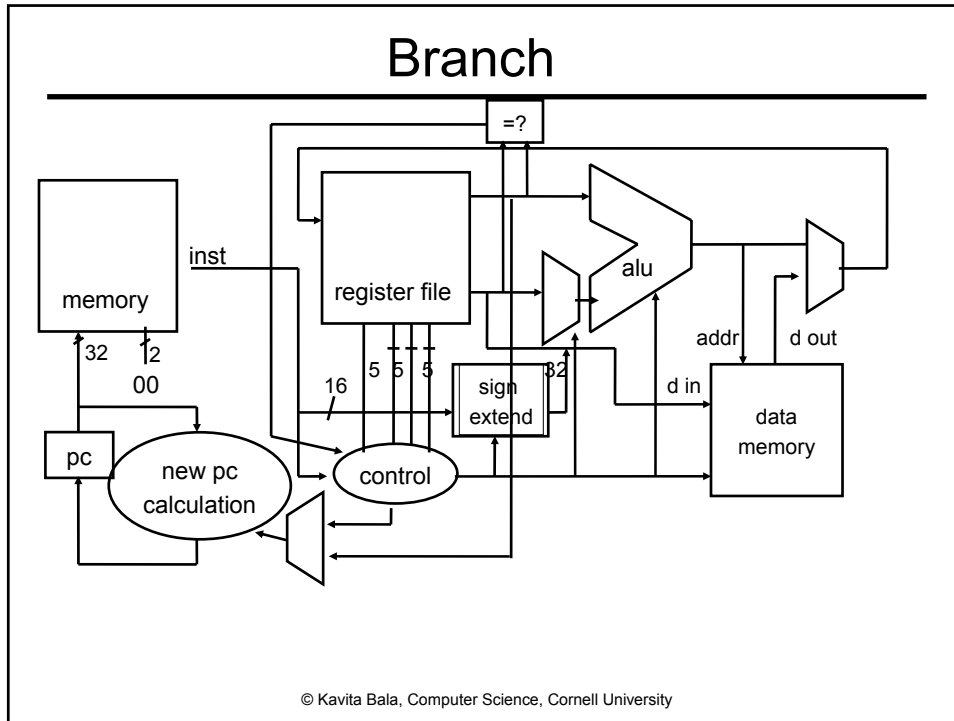
© Kavita Bala, Computer Science, Cornell University

Load



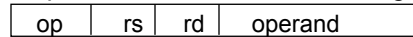
Store



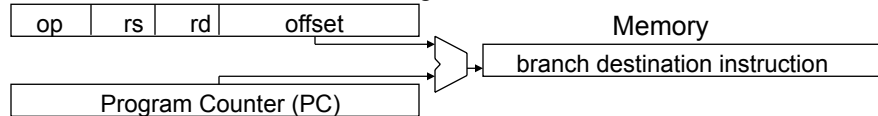


MIPS Addressing Modes

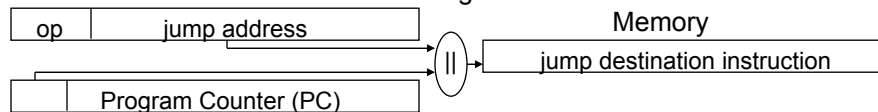
3. Operand: Immediate addressing



4. Instruction: PC-relative addressing



5. Instruction: Pseudo-direct addressing

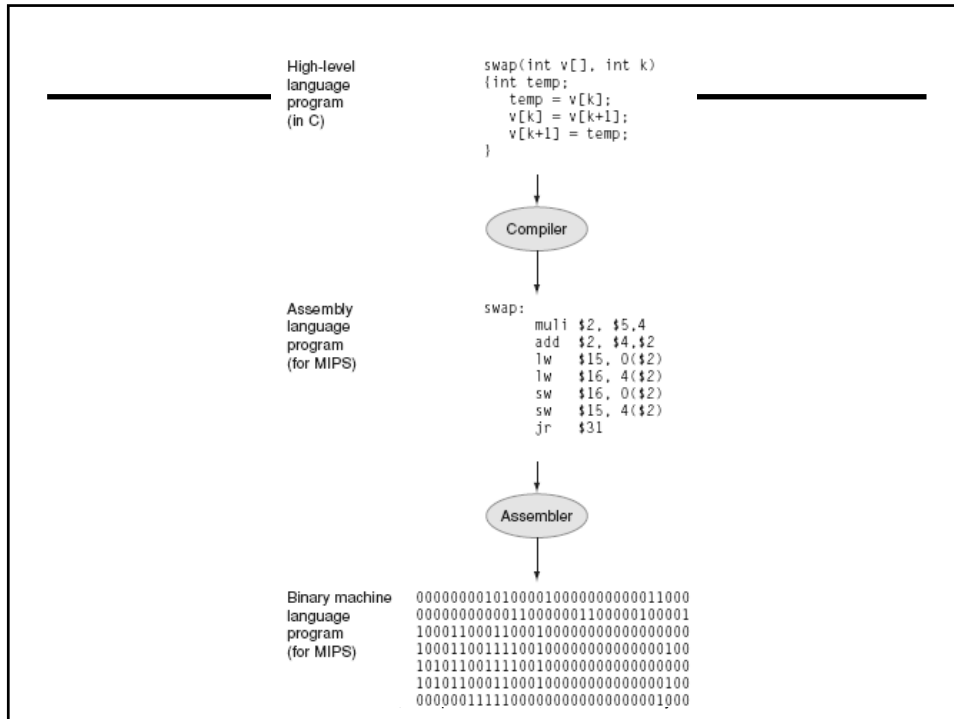


© Kavita Bala, Computer Science, Cornell University

Assembly Language Instructions

- Arithmetic
 - ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, SLT, SLTU
 - ADDI, ADDIU, ANDI, ORI, XORI, LUI, SLL, SRL, SLLV, SRLV, SRAV, SLTI, SLTIU
 - MULT, DIV, MFLO, MTLO, MFHI, MTHI
- Control Flow
 - BEQ, BNE, BLEZ, BLTZ, BGEZ, BGTZ
 - J, JR, JAL, JALR, BLTZAL, BGEZAL
- Memory
 - LW, LH, LB, LHU, LBU
 - SW, SH, SB
- Special
 - LL, SC, SYSCALL, BREAK, SYNC, COPROC

© Kavita Bala, Computer Science, Cornell University



Program Layout

- Programs consist of segments used for different purposes
 - Text: holds instructions
 - Data: holds statically allocated program data such as variables, strings, etc.

data

text

	“cornell cs”
	13
	25
	add r1,r2,r3
	ori r2, r4, 3
	...

Assembling Programs

```
.text
.ent main
main: la $4, Larray
      li $5, 15
      ...
      li $4, 0
      jal exit
.end main
.data
Larray:
      .long 51, 491, 3991
```

- Programs consist of a mix of instructions, pseudo-ops and assembler directives
- Assembler lays out binary values in memory based on directives

© Kavita Bala, Computer Science, Cornell University

Forward References

- Local labels can have forward references
- Two-pass assembly
 - Do a pass through the whole program, allocate instructions and lay out data, thus determining addresses
 - Do a second pass, emitting instructions and data, with the correct label offsets now determined

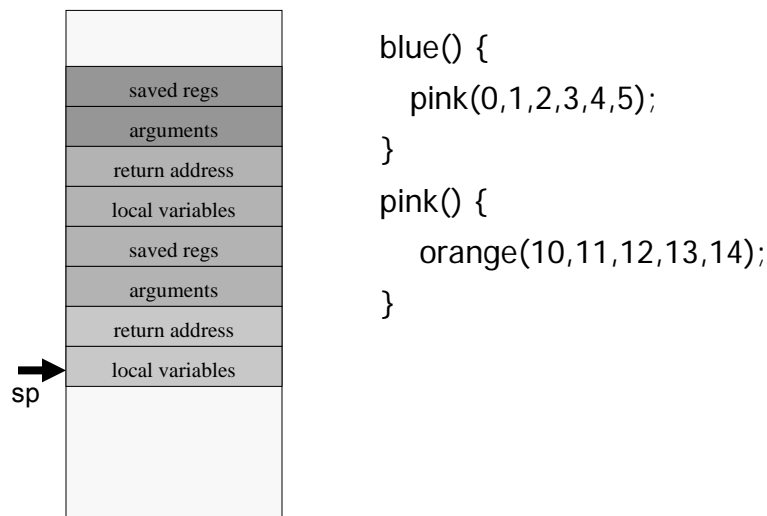
© Kavita Bala, Computer Science, Cornell University

Handling Forward References

- Example:
 - `bne $1, $2, L`
`sll $0, $0, 0`
`L: addiu $2, $3, 0x2`
- The assembler will change this to
 - `bne $1, $2, +1`
`sll $0, $0, 0`
`addiu $7, $8, $9`

© Kavita Bala, Computer Science, Cornell University

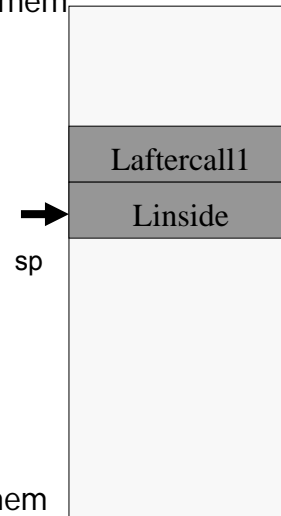
Frame Layout on Stack



© Kavita Bala, Computer Science, Cornell University

Call Stacks

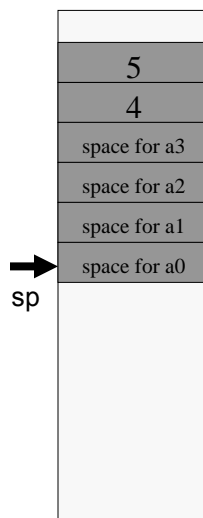
- A call stack contains activation records (aka stack frames) high mem
- Each activation record contains
 - the return address for that invocation
 - the local variables for that procedure



© Kavita Bala, Computer Science, Cornell University

Register Layout on Stack

```
main:
li a0, 0
li a1, 1
li a2, 2
li a3, 3
addiu sp,sp,-24
li $8, 4
sw $8, 16(sp)
li $8, 5
sw $8, 20(sp)
jal subf
// result in v0
```



- First four arguments are in registers
- The rest are on the stack
- There is room on the stack for the first four arguments, just in case

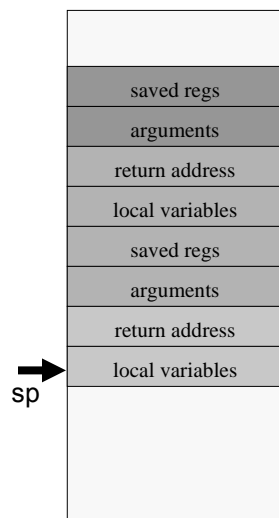
© Kavita Bala, Computer Science, Cornell University

Register Usage

- Callee-save
 - Save it if you modify it
 - Assumes caller needs it
 - Save the previous contents of the register on procedure entry, restore just before procedure return
 - E.g. \$31 (if you are a non-leaf... what is that?)
- Caller-save
 - Save it if you need it after the call
 - Assume callee can clobber any one of the registers
 - Save contents of the register before proc call
 - Restore after the call

© Kavita Bala, Computer Science, Cornell University

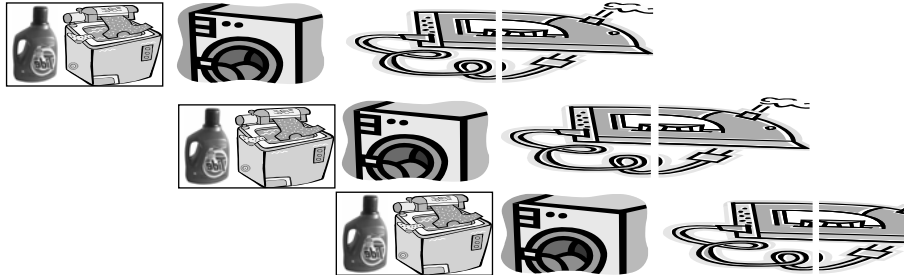
Buffer Overflows



```
blue() {  
    pink(0,1,2,3,4,5);  
}  
pink() {  
    orange(10,11,12,13,14);  
}  
orange() {  
    char buf[100];  
    gets(buf); // read string, no check  
}
```

© Kavita Bala, Computer Science, Cornell University

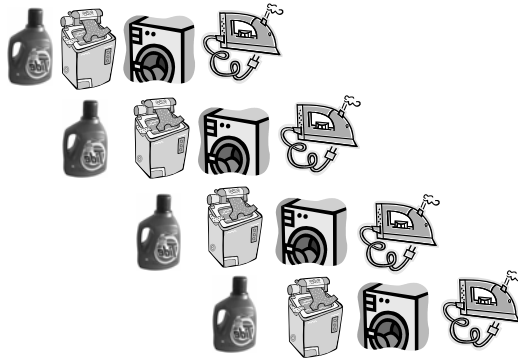
Pipelining



- Latency: ?
- Throughput: Batch every 45 minutes

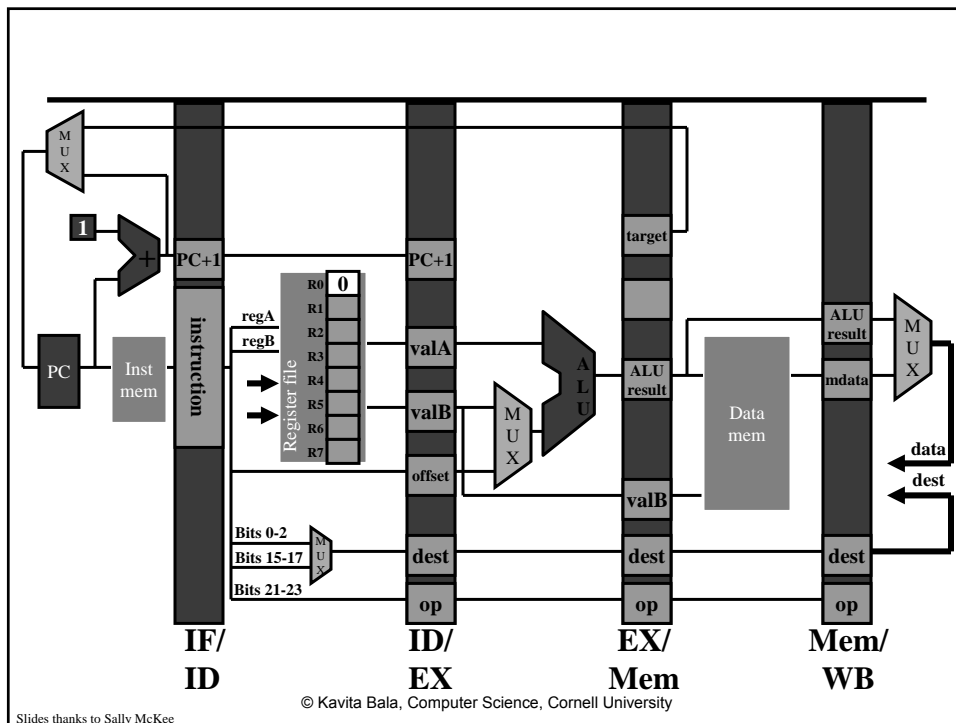
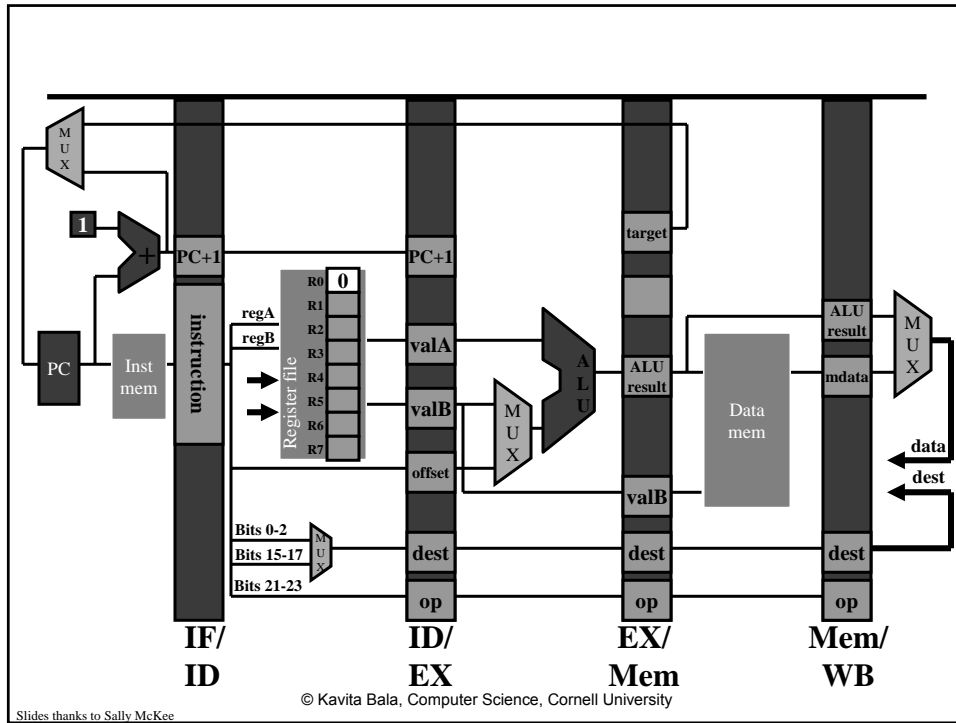
© Kavita Bala, Computer Science, Cornell University

Throughput is good



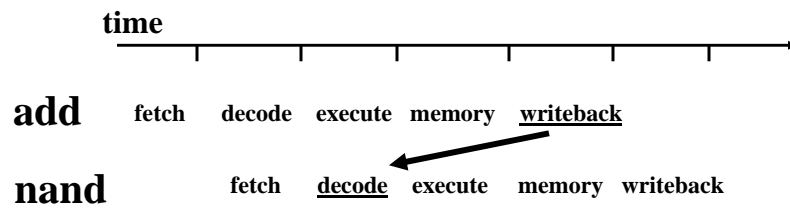
- What about latency?

© Kavita Bala, Computer Science, Cornell University



Data Hazards

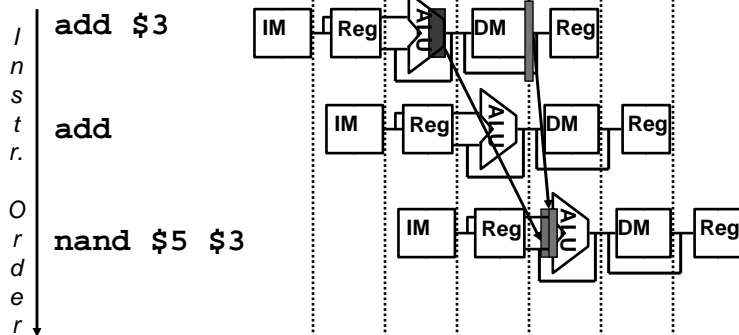
add 3 1 2
nand 5 3 4



If not careful, you read the wrong value of R3

© Kavita Bala, Computer Science, Cornell University

Forwarding Illustration

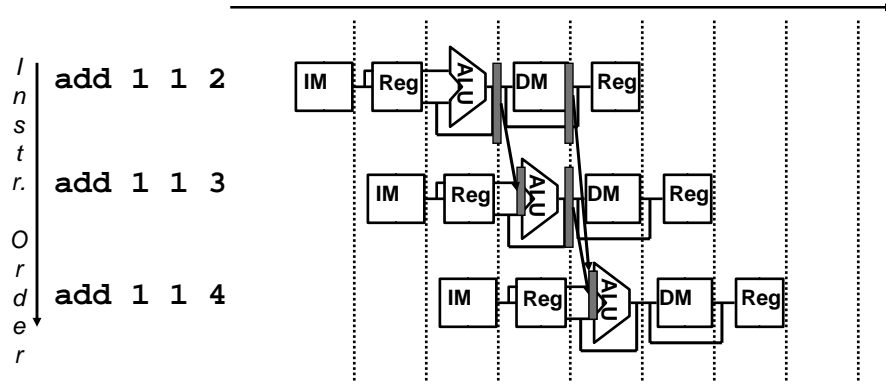


© Kavita Bala, Computer Science, Cornell University

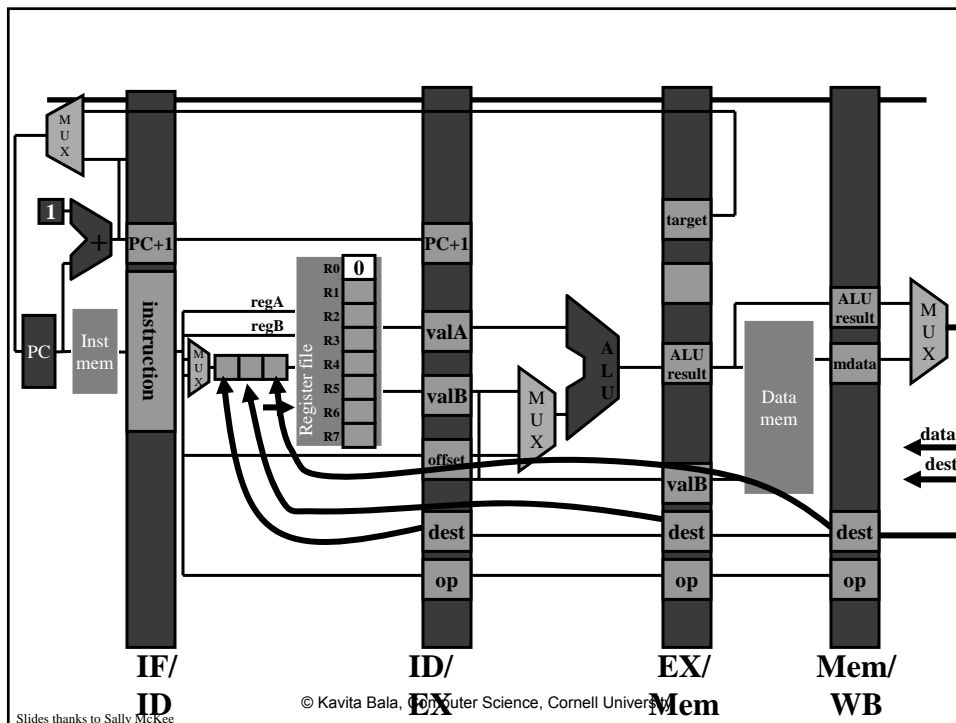
A tricky case

```

add 1 1 2
add 1 1 3
add 1 1 4
    
```



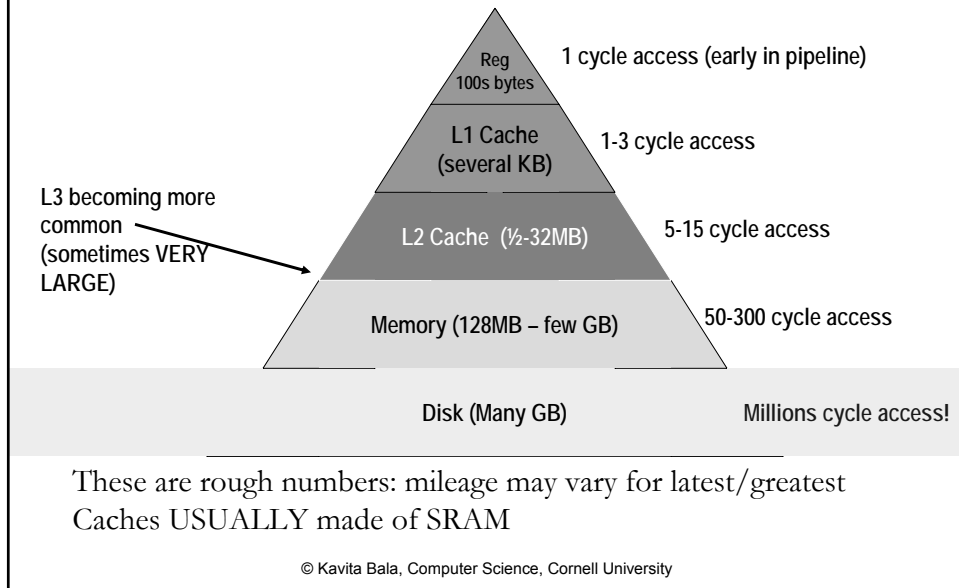
© Kavita Bala, Computer Science, Cornell University



Slides thanks to Sally McKee

© Kavita Bala, Computer Science, Cornell University

Cache Design 101



Misses

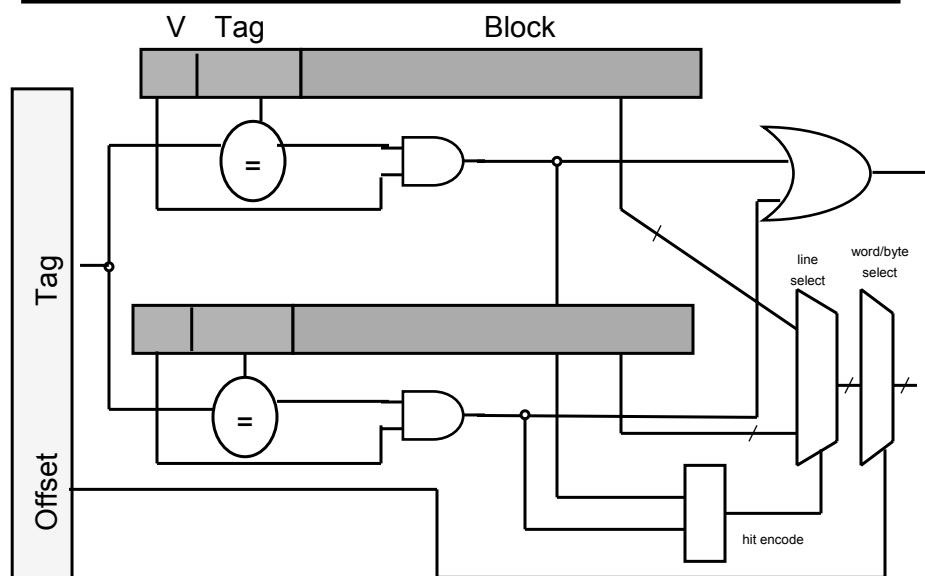
- Three types of misses
 - Cold
 - The line is being referenced for the first time
 - Capacity
 - The line was evicted because the cache was not large enough
 - Conflict
 - The line was evicted because of another access whose index conflicted

Direct Mapped Cache

- Simplest
- Block can only be in one line in the cache
- How to determine this location?
 - Use modulo arithmetic
 - (Block address) modulo (# cache blocks)
 - For power of 2, use log (cache size in blocks)

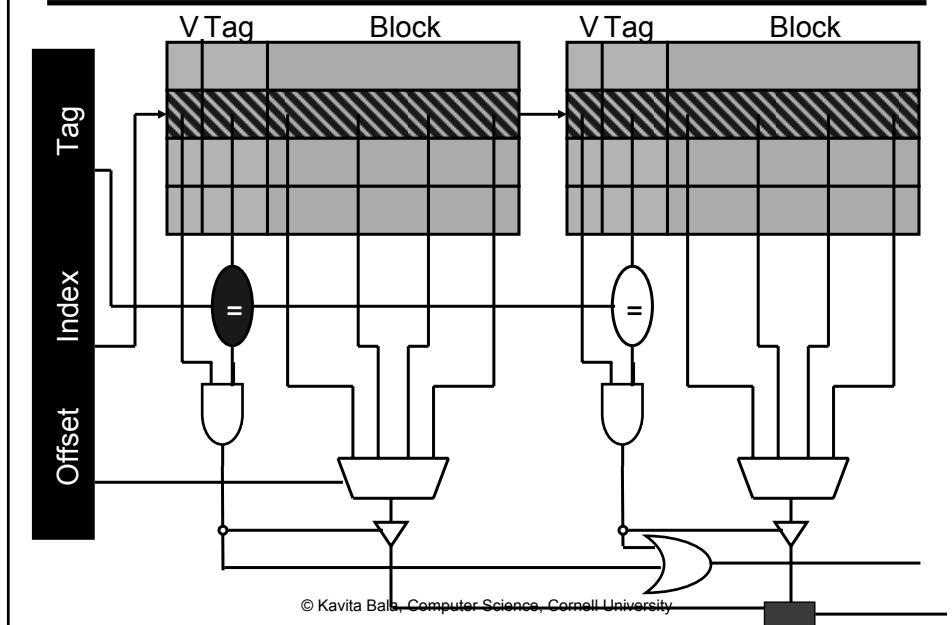
© Kavita Bala, Computer Science, Cornell University

Fully Associative Cache



© Kavita Bala, Computer Science, Cornell University

2-Way Set-Associative Cache

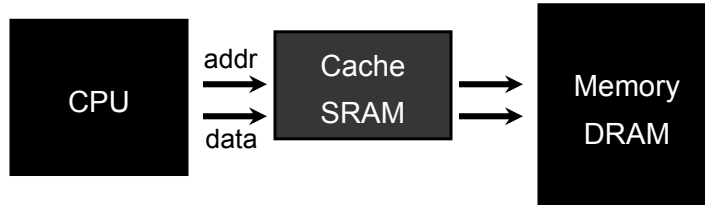


Eviction

- Which cache line should be evicted from the cache to make room for a new line?
 - Direct-mapped
 - no choice, must evict line selected by index
 - Associative caches
 - random: select one of the lines at random
 - round-robin: similar to random
 - FIFO: replace oldest line
 - LRU: replace line that has not been used in the longest time

© Kavita Bala, Computer Science, Cornell University

Cache Writes

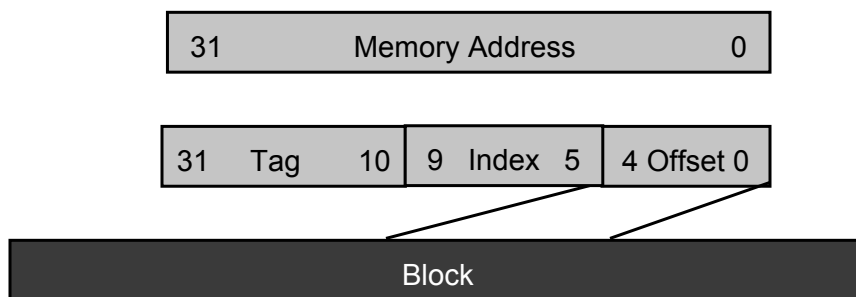


- No-Write
 - writes invalidate the cache and go to memory
- Write-Through
 - writes go to cache and to main memory
- Write-Back
 - write cache, write main memory only when block is evicted

© Kavita Bala, Computer Science, Cornell University

Tags and Offsets

- Tag: matching
- Offset: within block
- Valid bit: is the data valid?



© Kavita Bala, Computer Science, Cornell University

Short Performance Discussion

- Complicated
 - Time from start-to-end (wall-clock time)
 - System time, user time
 - CPI (Cycles per instruction)

- Ideal CPI?

© Kavita Bala, Computer Science, Cornell University

Cache Performance

- Consider hit (H) and miss ratio (M)
- $H \times AT_{\text{cache}} + M \times AT_{\text{memory}}$
- Hit rate = 1 – Miss rate
- Access Time is given in cycles
- Ratio of Access times, 1:50

- 90% : $.90 + .1 \times 50 = 5.9$
- 95% : $.95 + .05 \times 50 = .95+2.5=3.45$
- 99% : $.99 + .01 \times 50 = 1.49$
- 99.9%: $.999 + .001 \times 50 = 0.999 + 0.05 = 1.049$

© Kavita Bala, Computer Science, Cornell University

Cache Hit/Miss Rate

- Consider processor that is 2x times faster
 - But memory is same speed
- Since AT is access time in terms of cycle time: it doubles 2x
- $H \times AT_{\text{cache}} + M \times AT_{\text{memory}}$
- Ratio of Access times, 1:100
- $99\% : .99 + .01 \times 100 = 1.99$

© Kavita Bala, Computer Science, Cornell University

Cache Hit/Miss Rate

- Original is 1GHz, 1ns is cycle time
- CPI (cycles per instruction): 1.49
- Therefore, 1.49 ns for each instruction

- New is 2GHz, 0.5 ns is cycle time.
- CPI: 1.99, 0.5ns. 0.995 ns for each instruction.

- So it doesn't go to 0.745 ns for each instruction.
- Speedup is 1.5x (not 2x)

© Kavita Bala, Computer Science, Cornell University

Cache Conscious Programming

```
int a[NCOL][NROW];
int sum = 0;

for(j = 0; j < NCOL; ++j)
    for(i = 0; i < NROW; ++i)
        sum += a[j][i];
```

1	11									
2	12									
3	13									
4	14									
5	15									
6										
7										
8										
9										
10										

- Every access is a cache miss!

© Kavita Bala, Computer Science, Cornell University

Cache Conscious Programming

```
int a[NCOL][NROW];
int sum = 0;

for(i = 0; i < NROW; ++i)
    for(j = 0; j < NCOL; ++j)
        sum += a[j][i];
```

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15					

- Same program, trivial transformation, 3 out of four accesses hit in the cache

© Kavita Bala, Computer Science, Cornell University

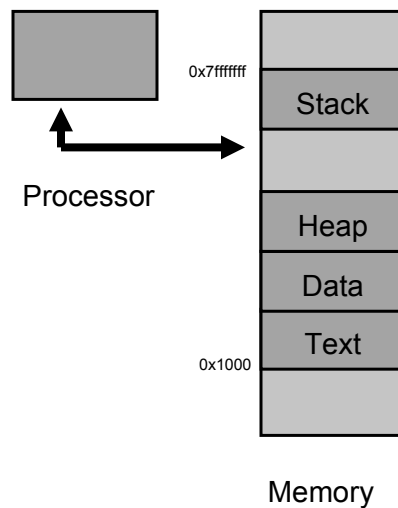
Can answer the question.....

- A: for i = 0 to 99
 - for j = 0 to 999
 - $A[i][j] = \text{complexComputation}()$
- B: for j = 0 to 999
 - for i = 0 to 99
 - $A[i][j] = \text{complexComputation}()$
- Why is B 15 times slower than A?

© Kavita Bala, Computer Science, Cornell University

Processor & Memory

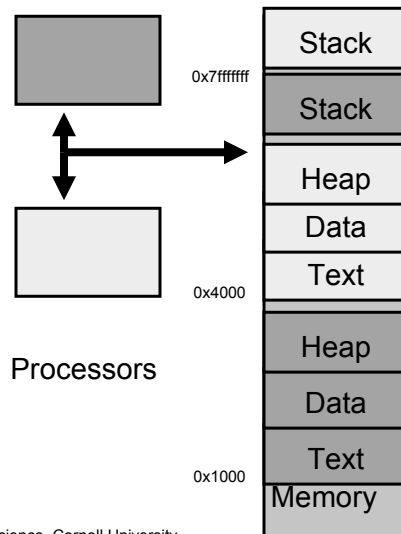
- Currently, the processor's address lines are directly routed via the system bus to the memory banks
 - Simple, fast
- What happens when the program issues a load or store to an invalid location?
 - e.g. $0x00000000$?
 - uninitialized pointer



© Kavita Bala, Computer Science, Cornell University

Physical Addressing Problems

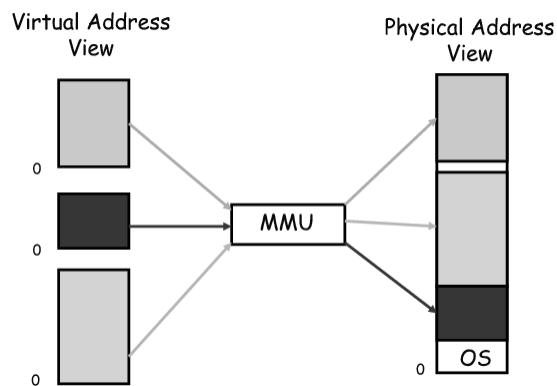
- What happens when another program is executed concurrently on another processor?
 - The addresses will conflict
- We could try to relocate the second program to another location
 - Assuming there is one
 - Introduces more problems!



© Kavita Bala, Computer Science, Cornell University

Address Space

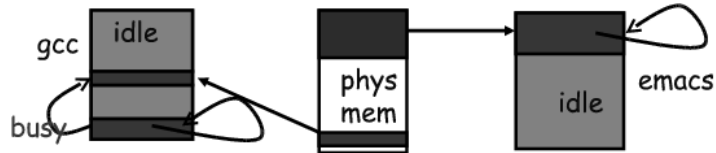
- Memory Management Unit (MMU)
 - Combination of hardware and software



© Kavita Bala, Computer Science, Cornell University

Virtual Memory Advantages

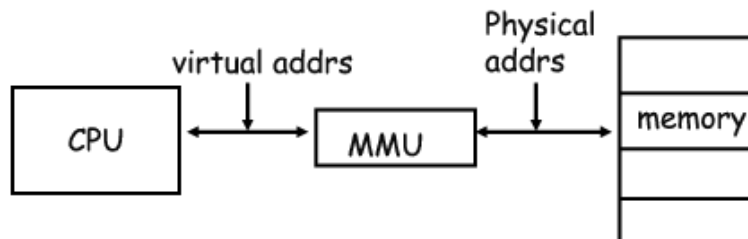
- Can relocate program while running
- Virtualization
 - In CPU: if process is not doing anything, switch
 - In memory: when not using it, somebody else can use it



© Kavita Bala, Computer Science, Cornell University

How to make it work?

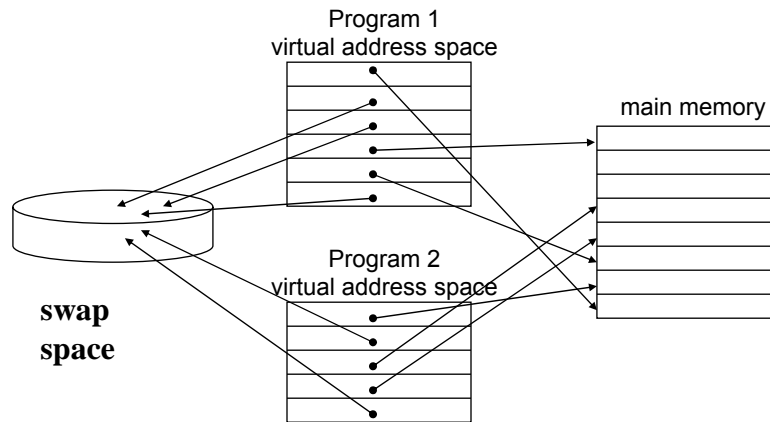
- Challenge: Virtual Memory can be slow!
- At run-time: virtual address must be translated to a physical address
- MMU (combination of hardware and software)



© Kavita Bala, Computer Science, Cornell University

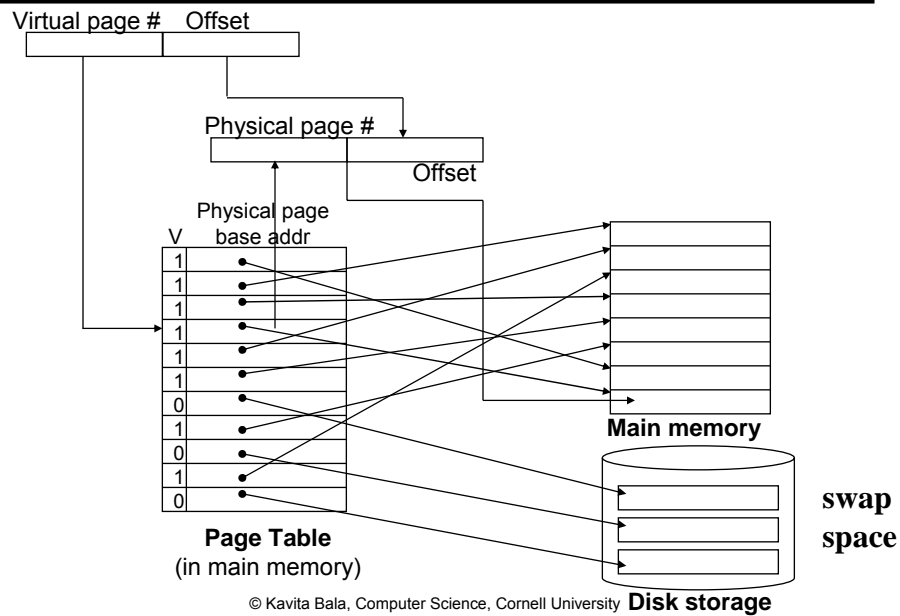
Two Programs Sharing Physical Memory

- The starting location of each page (either in main memory or in secondary memory) is contained in the program's page table



© Kavita Bala, Computer Science, Cornell University

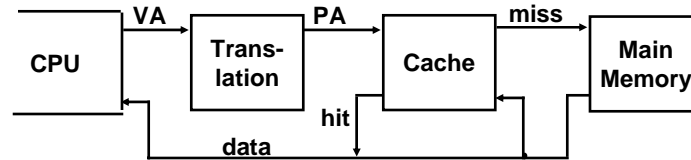
Page Table for Translation



© Kavita Bala, Computer Science, Cornell University

Virtual Addressing with a Cache

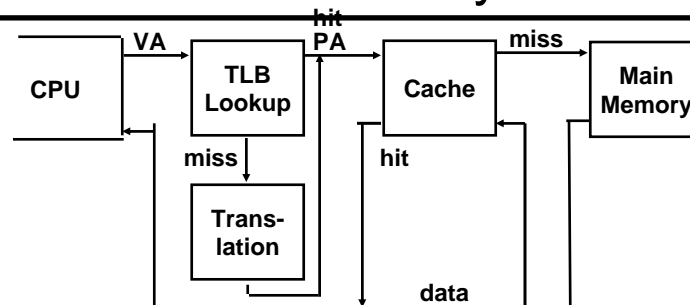
- Thus it takes an *extra* memory access to translate a VA to a PA



- This makes memory (cache) accesses very expensive (if every access was really *two* accesses)

© Kavita Bala, Computer Science, Cornell University

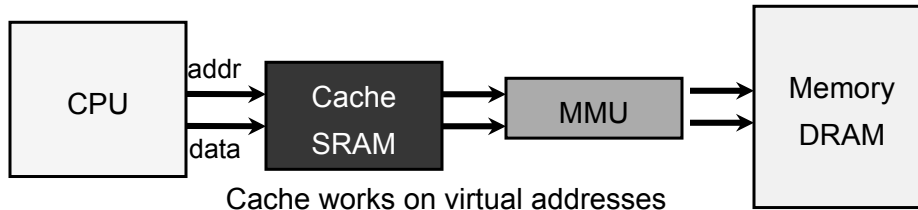
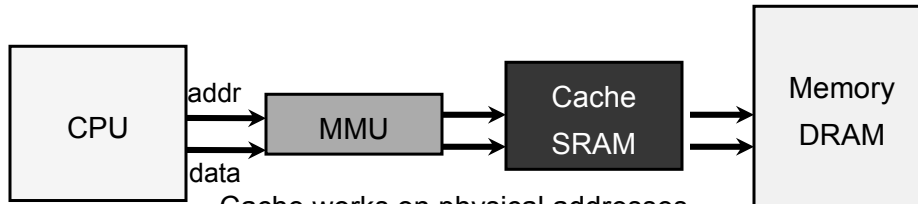
A TLB in the Memory Hierarchy



- A TLB miss:
 - If the page is not in main memory, then it's a true page fault
 - Takes 1,000,000's of cycles to service a page fault
- TLB misses are much more frequent than true page faults

© Kavita Bala, Computer Science, Cornell University

Virtual vs. Physical Caches

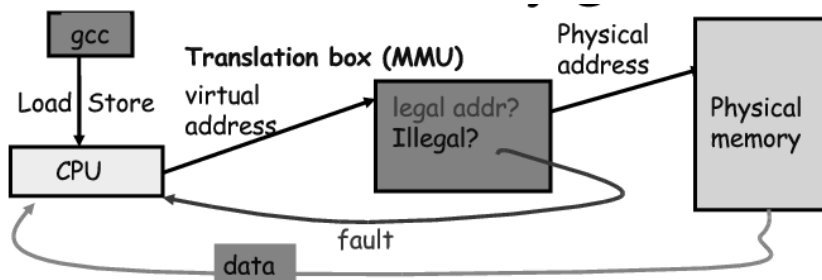


- L1 (on-chip) caches are typically virtual
- L2 (off-chip) caches are typically physical

© Kavita Bala, Computer Science, Cornell University

Address Translation

- Translation is done through the page table
 - A virtual memory miss (i.e., when the page is not in physical memory) is called a page fault



© Kavita Bala, Computer Science, Cornell University

Hardware/Software Boundary

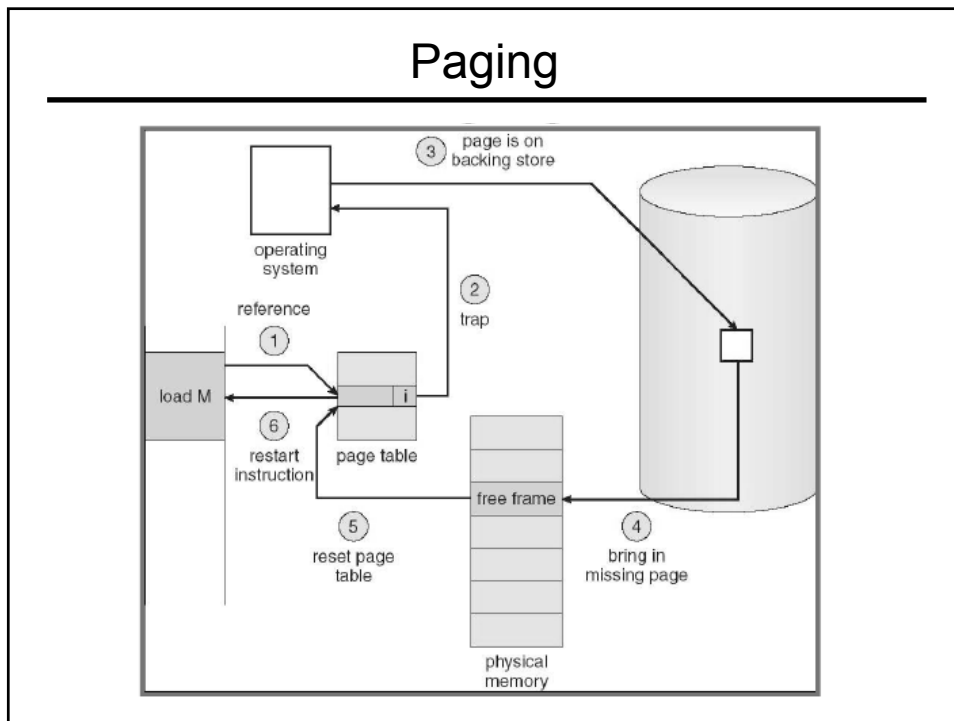
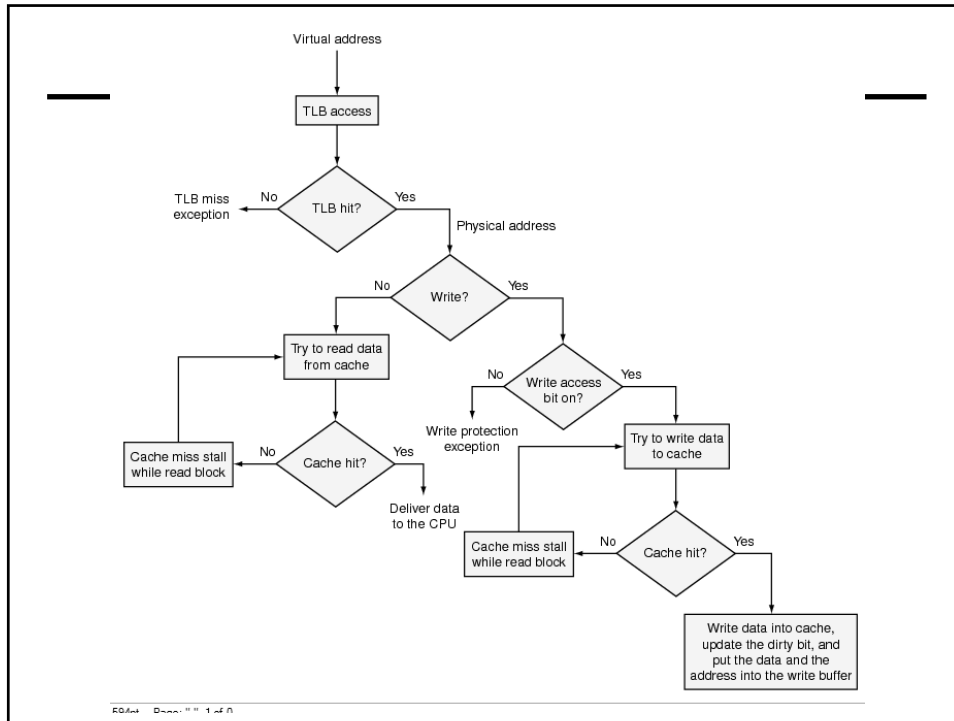
- Virtual to physical address translation is assisted by hardware?
 - Translation Lookaside Buffer (TLB) that caches the recent translations
 - TLB access time is part of the cache hit time
 - May allot an extra stage in the pipeline for TLB access
 - TLB miss
 - Can be in software (kernel handler) or hardware

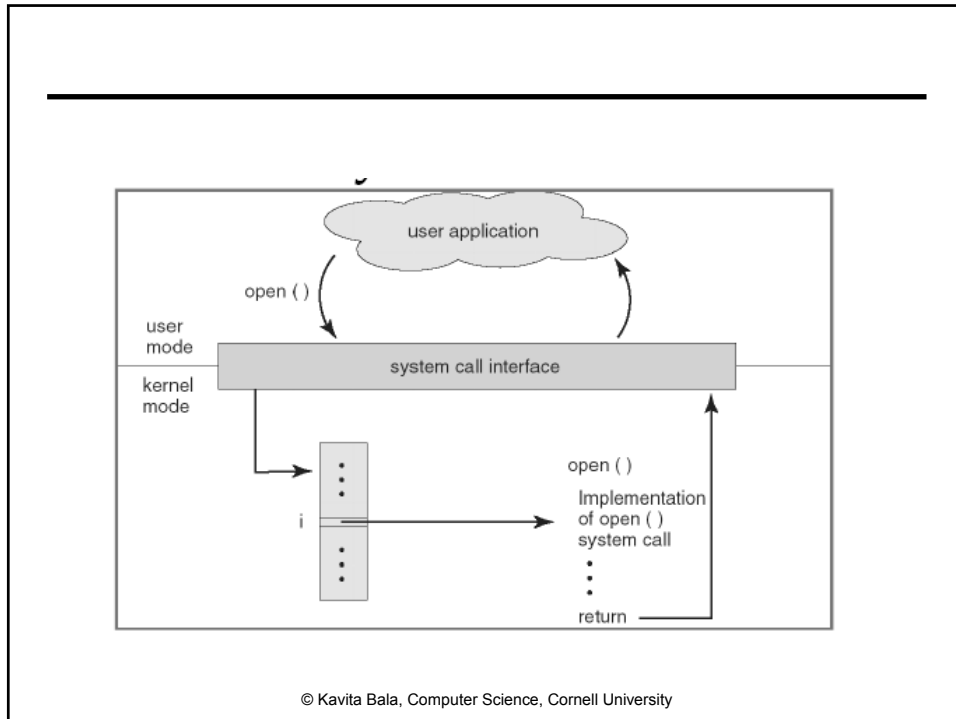
© Kavita Bala, Computer Science, Cornell University

Hardware/Software Boundary

- Virtual to physical address translation is assisted by hardware?
 - Page table storage, fault detection and updating
 - Page faults result in interrupts (precise) that are then handled by the OS
 - Hardware must support (i.e., update appropriately) Dirty and Reference bits (e.g., ~LRU) in the Page Tables

© Kavita Bala, Computer Science, Cornell University





Exceptions

- System calls are control transfers to the OS, performed under the control of the user program
- Sometimes, need to transfer control to the OS at a time when the user program least expects it
 - Division by zero,
 - Alert from power supply that electricity is going out,
 - Alert from network device that a packet just arrived,
 - Clock notifying the processor that clock just ticked
- Some of these causes for interruption of execution have nothing to do with the user application
- Need a (slightly) different mechanism, that allows resuming the user application

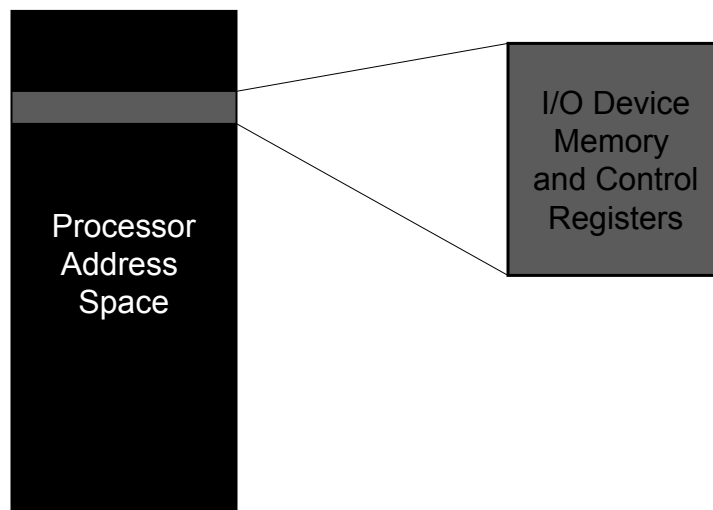
© Kavita Bala, Computer Science, Cornell University

Terminology

- Trap
 - Any kind of a control transfer to the OS
- Syscall
 - Synchronous, program-initiated control transfer from user to the OS to obtain service from the OS
 - e.g. SYSCALL
- Exception
 - Asynchronous, program-initiated control transfer from user to the OS in response to an exceptional event
 - e.g. Divide by zero
- Interrupt
 - Asynchronous, device-initiated control transfer from user to the OS
 - e.g. Clock tick, network packet

© Kavita Bala, Computer Science, Cornell University

Memory-Mapped I/O



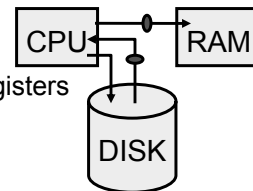
© Kavita Bala, Computer Science, Cornell University

DMA: Direct Memory Access

◆ Non-DMA transfer: I/O device \leftrightarrow CPU \leftrightarrow RAM

– for ($i = 1 \dots n$)

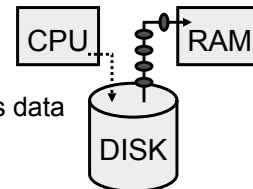
- CPU sends transfer request to device
- I/O writes data to bus, CPU reads into registers
- CPU writes data to registers to memory



◆ DMA transfer: I/O device \leftrightarrow RAM

– CPU sets up DMA request on device
– for ($i = 1 \dots n$)

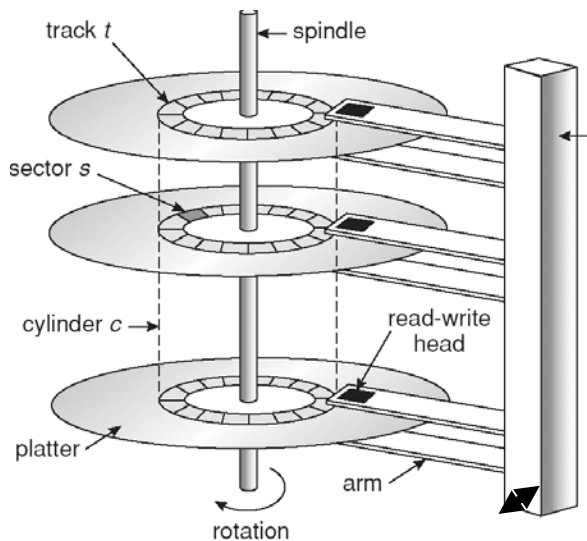
- I/O device writes data to bus, RAM reads data



Based on lecture from Kevin Walsh

© Kavita Bala, Computer Science, Cornell University

Disk Physics



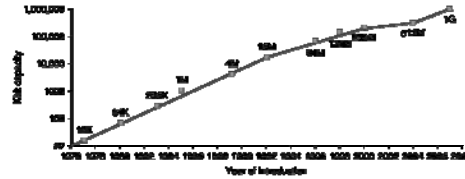
Typical parameters :

- 1 spindle
- 1 arm assembly
- 1-4 platters
- 1-2 sides/platter
- 1 head per side (but only 1 active head at a time)
- 4,200 – 15,000 RPM

© Kavita Bala, Computer Science, Cornell University

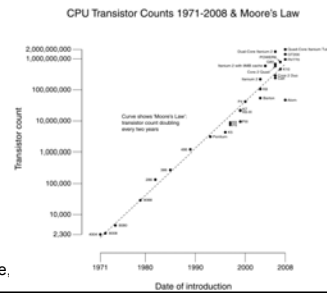
Technology Trends

- DRAM capacity
 - Increased
 - Reduced cost



DRAM capacity

- Moore's Law
 - Speed?
 - Not really

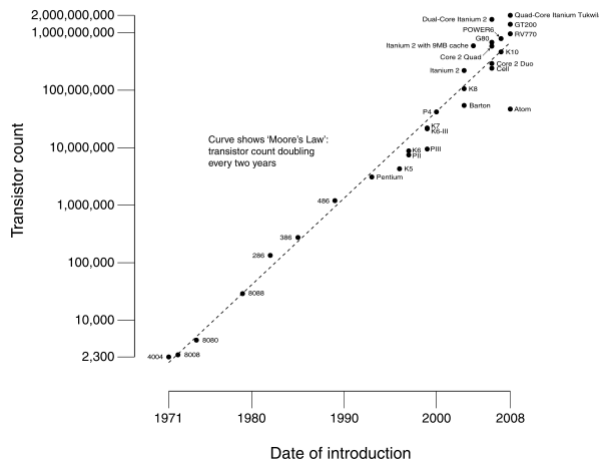


© Kavita Bala, Computer Science,

Moore's Law

- Law about transistor count

CPU Transistor Counts 1971-2008 & Moore's Law



Review: Performance Summary

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
 - Algorithm: affects IC, possibly CPI
 - Programming language: affects IC, CPI
 - Compiler: affects IC, CPI
 - Instruction set architecture: affects IC, CPI, T_c

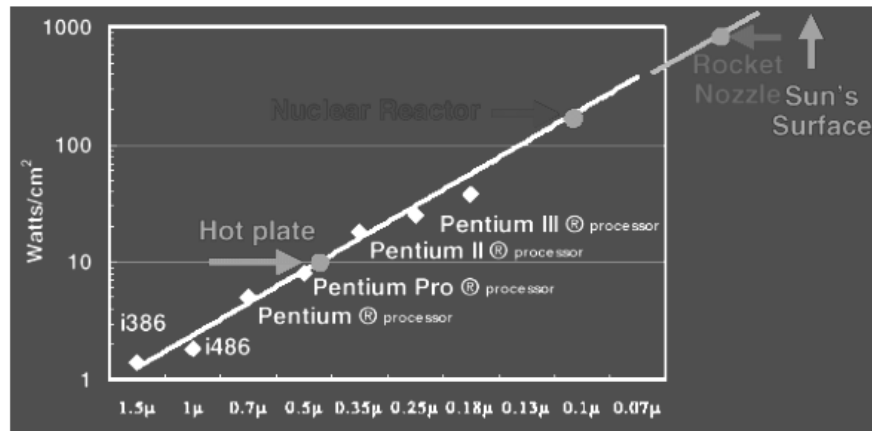
© Kavita Bala, Computer Science, Cornell University

Why Multicore?

- Moore's law
 - A law about transistors
 - Smaller means faster transistors
- Power consumption growing with transistors

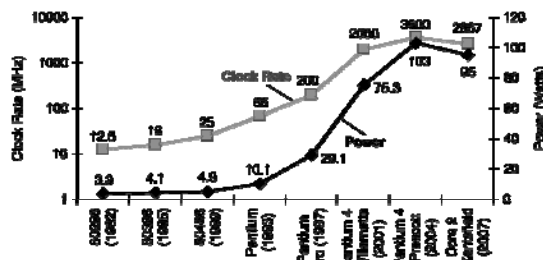
© Kavita Bala, Computer Science, Cornell University

Power Limits Performance



© Kavita Bala, Computer Science, Cornell University

Power Trends



- In CMOS IC technology

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

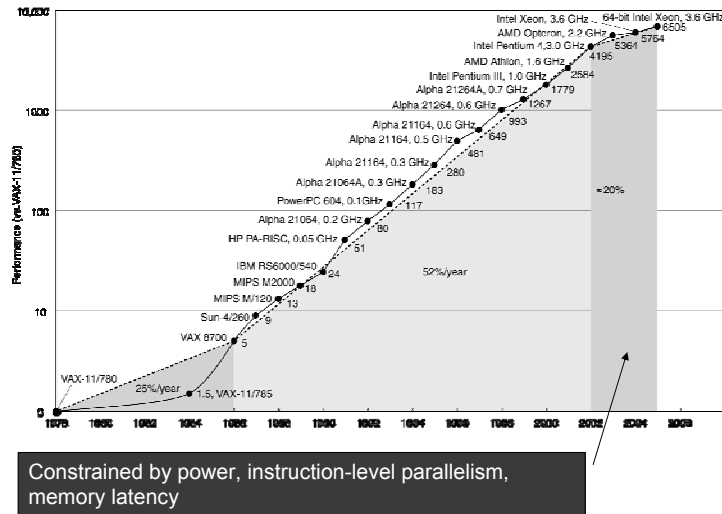
×30

5V → 1V

×1000

© Kavita Bala, Computer Science, Cornell University

Uniprocessor Performance



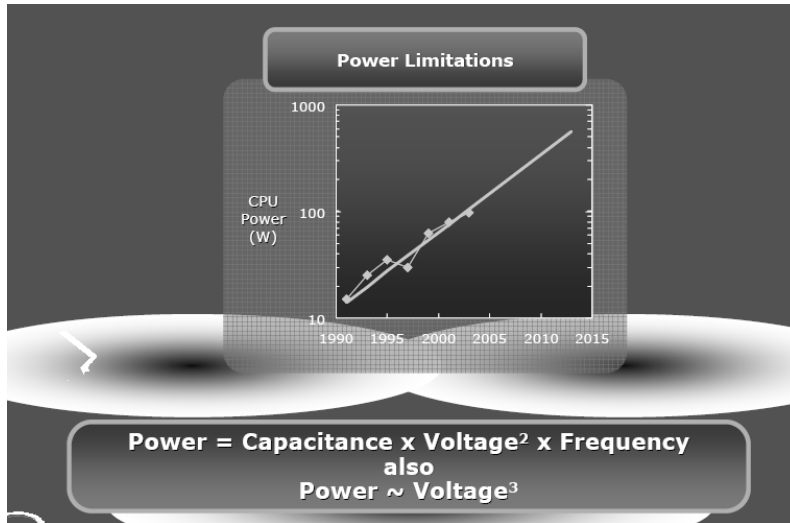
© Kavita Bala, Computer Science, Cornell University

Why Multicore?

- Moore's law
 - A law about transistors
 - Smaller means faster transistors
- Power consumption growing with transistors
- The power wall
 - We can't reduce voltage further
 - We can't remove more heat
- How else can we improve performance?

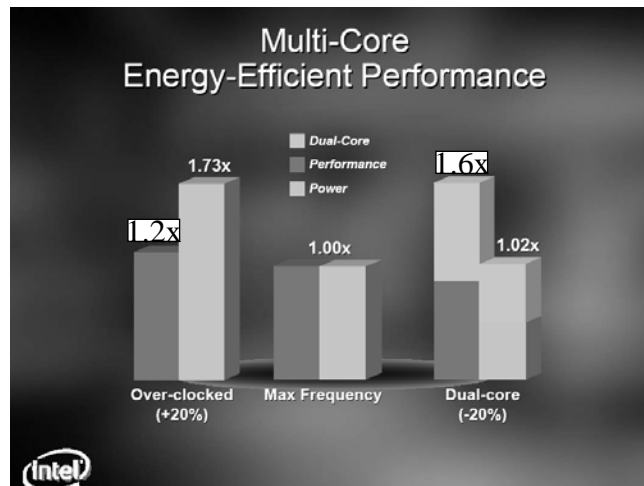
© Kavita Bala, Computer Science, Cornell University

Intel's argument



© Kavita Bala, Computer Science, Cornell University

Multi-Core Energy-Efficient Performance



© Kavita Bala, Computer Science, Cornell University

Amdahl's Law

- Task: serial part, parallel part
- As number of processors increases,
 - time to execute parallel part goes to zero
 - time to execute serial part remains the same
- *Serial part eventually dominates*
- Must parallelize ALL parts of task

$$\text{Speedup}(E) = \frac{\text{Execution Time without } E}{\text{Execution Time with } E}$$

© Kavita Bala, Computer Science, Cornell University

Amdahl's Law

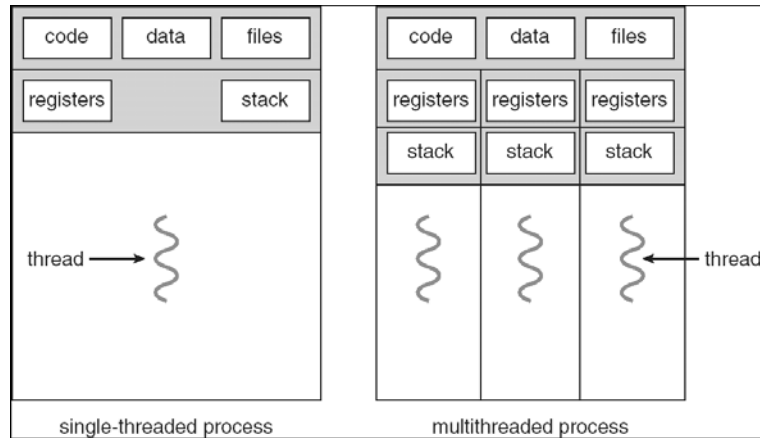
- Consider an improvement E
- F of the execution time is affected
- S is the speedup

Execution time (with E) = $((1 - F) + F/S) \cdot$ Execution time (without E)

$$\text{Speedup (with } E) = \frac{1}{(1 - F) + F/S}$$

© Kavita Bala, Computer Science, Cornell University

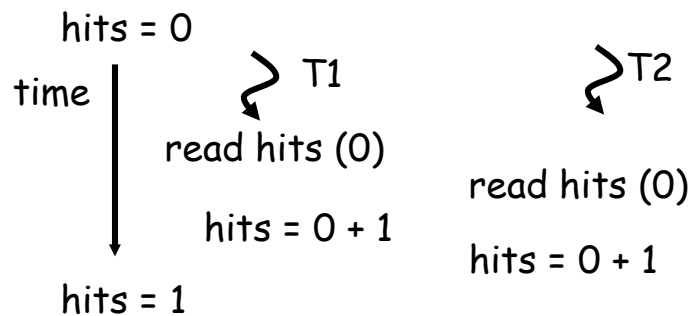
Multithreaded Processes



© Kavita Bala, Computer Science, Cornell University

Shared counters

- Usual result: works fine.
- Possible result: lost update!



- Occasional timing-dependent failure \Rightarrow Difficult to debug
- Called a *race condition*

© Kavita Bala, Computer Science, Cornell University

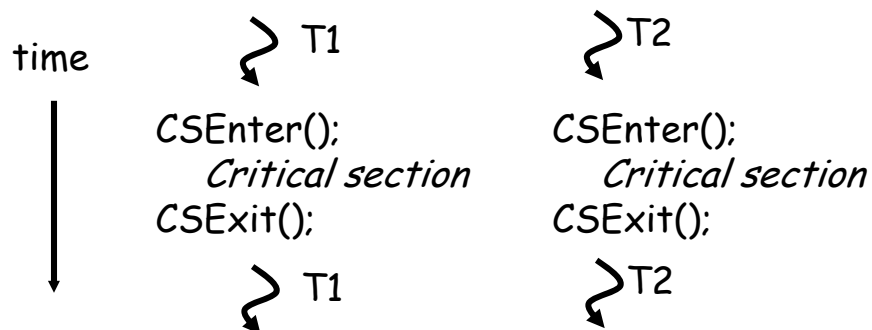
Race conditions

- Def: a timing dependent error involving shared state
 - Whether it happens depends on how threads scheduled: who wins “races” to instructions that update state
 - Races are intermittent, may occur rarely
 - Timing dependent = small changes can hide bug
 - A program is correct *only* if *all possible* schedules are safe
 - Number of possible schedule permutations is huge
 - Need to imagine an adversary who switches contexts at the worst possible time

© Kavita Bala, Computer Science, Cornell University

Critical Sections

- Basic way to eliminate races: use *critical sections* that only one thread can be in
 - Contending threads must wait to enter



© Kavita Bala, Computer Science, Cornell University

Mutexes

- Critical sections typically associated with mutual exclusion locks (*mutexes*)
- Only one thread can hold a given mutex at a time
- Acquire (lock) mutex on entry to critical section
 - Or block if another thread already holds it
- Release (unlock) mutex on exit
 - Allow one waiting thread (if any) to acquire & proceed

```
pthread_mutex_init(m);  
pthread_mutex_lock(m);    pthread_mutex_lock(m);  
    hits = hits+1;        hits = hits+1;  
pthread_mutex_unlock(m); pthread_mutex_unlock(m);
```



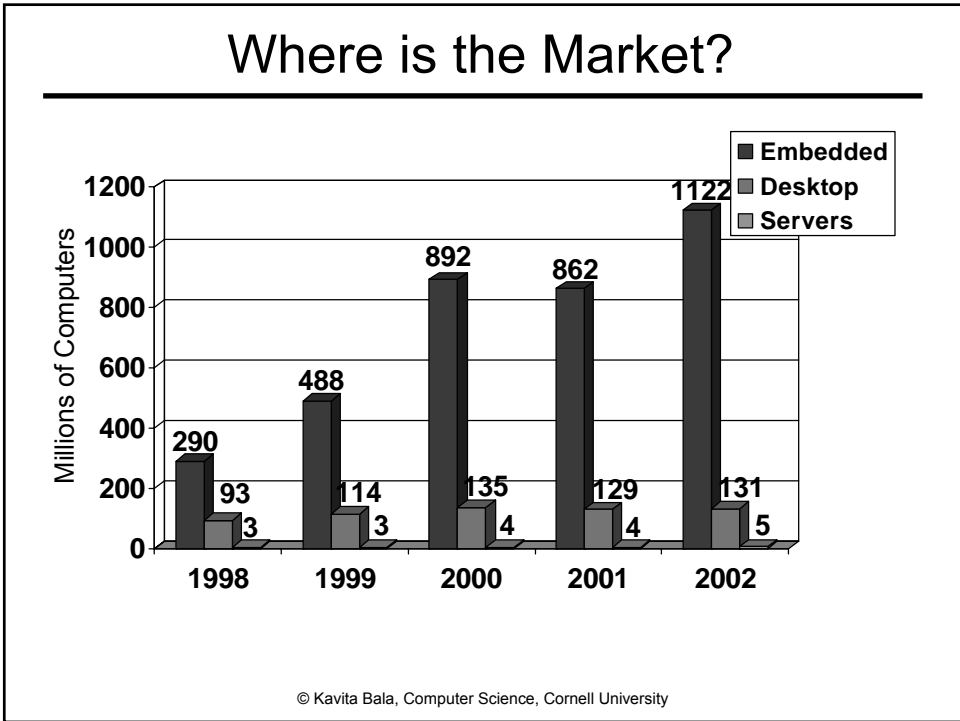
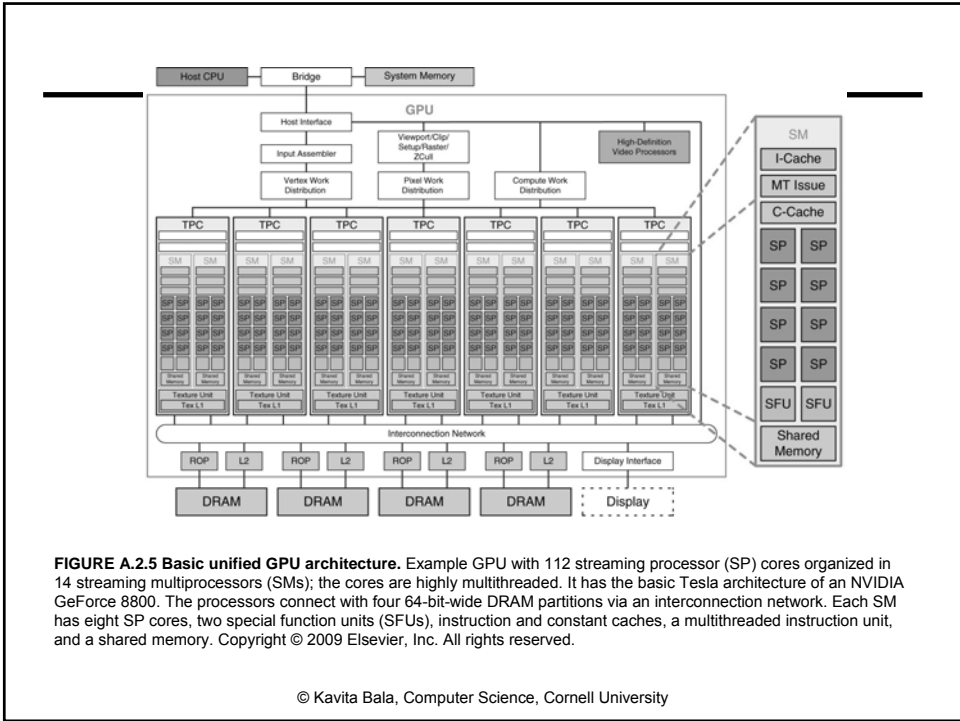
© Kavita Bala, Computer Science, Cornell University

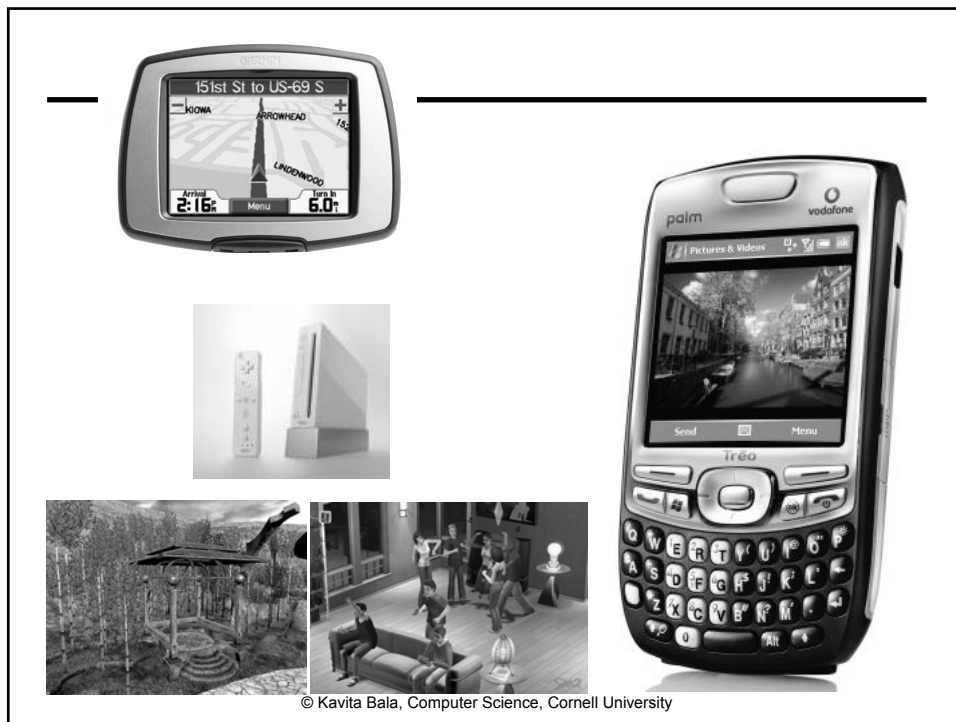
Protecting an invariant

```
// invariant: data is in buffer[first..last-1]. Protected by m.  
pthread_mutex_t *m;  
char buffer[1000];  
int first = 0, last = 0;  
  
void put(char c) {  
    pthread_mutex_lock(m);  
    buffer[last] = c;  
    last++;  
    pthread_mutex_unlock(m);  
}  
  
char get() {  
    pthread_mutex_lock(m);  
    char c = buffer[first];  
    first++;      X what if first==last?  
    pthread_mutex_unlock(m);  
}
```

- Rule of thumb: all updates that can affect invariant become critical sections.

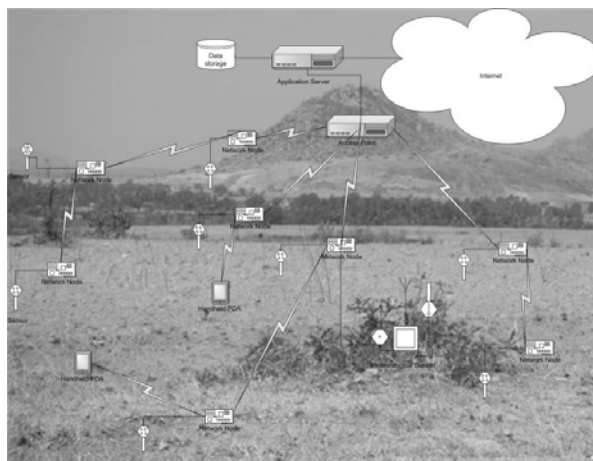
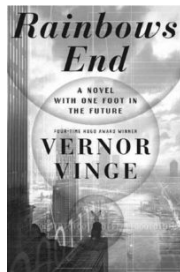
© Kavita Bala, Computer Science, Cornell University





Where to?

- Smart Dust....



© Kavita Bala, Computer Science, Cornell University

Where to?

- CS 3110: Better concurrent programming
- CS 4410: The Operating System!
- CS 4450: Networking
- CS 6620: Graphics
- And many more...

© Kavita Bala, Computer Science, Cornell University

Thank you!

© Kavita Bala, Computer Science, Cornell University