Number Rep and Floating Point

Ki-10 Gi-30 Pebi-50 Zebi-70
Mi-20 Ti-40 Exbi-60 Yobi-80

Signmag: one sign bit, rest same as unsigned
Biased: predetermined bias, applied to all numbers
One's complement: flip bits for negative
Two's complement: flip bits and add 1

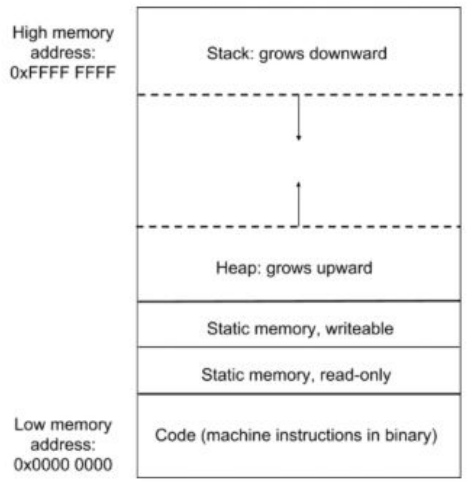
Interpretation	Range	Lowest	Highest	# of Representable Numbers
Unsigned	[0, 2^n - 1]	00...00	11...11	2^n
Sign and Magnitude	[-(2^{n-1} - 1), 2^{n-1} - 1]	11...11	01...11	2^n - 1
One's Complement	[-(2^{n-1} - 1), 2^{n-1} - 1]	10...00	01...11	2^n - 1
Two's Complement	[-2^{n-1}, 2^{n-1} - 1]	10...00	01...11	2^n
Biased	[bias, 2^n - 1 + bias]	00...00	11...11	2^n

Bias: $2^{(\#\text{ of exponent bits } -1)} - 1$

Minimum unrepresentable int is $2^{mantissa+1} + 1$
Multiply $2^{-mantissa} * 2^{exp-bias}$ to find step size
Two's compement -1 is 1111111

C

Char - 1B ALWAYS, everything else depends on system
8bits = 1 byte
Pointers same size throughout system
Pointer arithmetic: increment by sizeof(type_pointing_to), *not* 1
Stack: function local variables, strings allocated as ARRAYS, pointers //String literals put in read-only memory is feature of compiler not language // char c[40] = "fjdkljsl\n"
Heap: dynamically allocated memory (with malloc etc)
Static: global variables, statically allocated strings- char *c = "hi"
Code: machine instructions currently being run



1	8	23
Sign	Exponent	Mantissa/Significand/Fraction

For normalized floats:

$$\textbf{Value} = (-1)^{Sign} * 2^{Exp-Bias} * 1.\textbf{significand}_2$$

For denormalized floats:

$$\textbf{Value} = (-1)^{Sign} * 2^{Exp-Bias+1} * 0.\textbf{significand}_2$$

Exponent	Significand	Meaning
0	Anything	Denorm
1-254	Anything	Normal
255	0	Infinity
255	Nonzero	NaN

CALL

Compiler (.c → .s)
- May contain pseudoinstructions

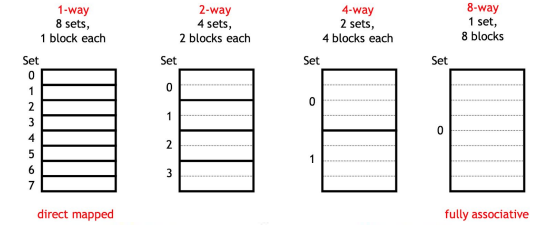
Assembler (.s → .o)
- Create machine language
- Has directives
- Replaces pseudo instructions
- Two passes:
  1. Replace pseudo instructions
  2. Generate relative obj
- Set up info for linker
- Creates symbol table: keeps track of labels and data for OTHER files
- Relocation table: keeps track of addresses from other files that are used by THIS file

Linker (.o → executable, a.out [default])
- Link file texts, data, relocate necessary addresses
- Update and fill in absolute addresses that Assembler looked at

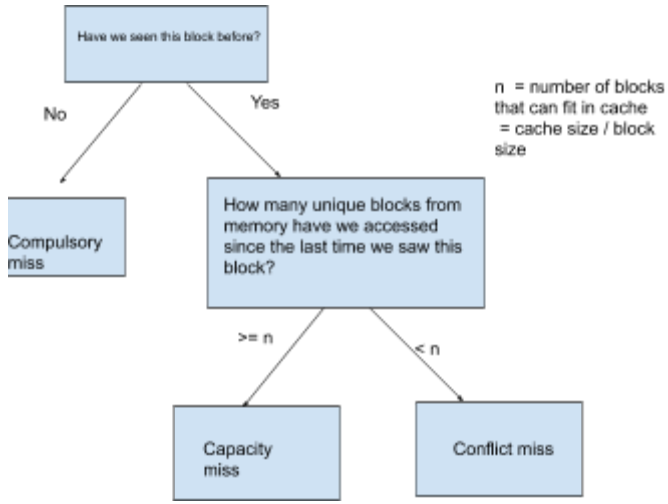
Loader (executable → run!)
- Load executables into memory and run
- Create new address space
- Copies instructions
- Copies args in program to stack
- Initialize machine registers

SDS

t_hold ≤ t_shortestpath = CLK_to_Q + time_CL
t_longest_path + t_setup ≤ t_clk_cycle
Critical path = max delay = Clk_to_Q + time_CL + setup time
Min clock period ≥ critical path
Max frequency = 1/min_period
Two types of circuits:
1. Combinational logic
2. State elements



Name	AND Form	OR form
Commutative	$AB = BA$	$A + B = B + A$
Associative	$AB(C) = A(BC)$	$A + (B + C) = (A + B) + C$
Identity	$1A = A$	$0 + A = A$
Null	$0A = 0$	$1 + A = 1$
Absorption	$A(A + B) = A$	$A + AB = A$
Distributive	$(A + B)(A + C) = A + BC$	$A(B + C) = AB + AC$
Idempotent	$A(A) = A$	$A + A = A$
Inverse	$A(\overline{A}) = 0$	$A + \overline{A} = 1$
Demorgan's	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}(\overline{B})$



Control Logic Truth Table

Inst[31:0]	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	+4	*	*	Reg	Reg	Add	Read	1	ALU
sub	*	*	+4	*	*	Reg	Reg	Sub	Read	1	ALU
(R-R Op)	*	*	+4	*	*	Reg	Reg	(Op)	Read	1	ALU
addi	*	*	+4	I	*	Reg	Imm	Add	Read	1	ALU
lw	*	*	+4	I	*	Reg	Imm	Add	Read	1	Mem
sw	*	*	+4	S	*	Reg	Imm	Add	Write	0	*
beq	0	*	+4	B	*	PC	Imm	Add	Read	0	*
beq	1	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	0	*	ALU	B	*	PC	Imm	Add	Read	0	*
bne	1	*	+4	B	*	PC	Imm	Add	Read	0	*
blt	*	1	ALU	B	0	PC	Imm	Add	Read	0	*
bltu	*	1	ALU	B	1	PC	Imm	Add	Read	0	*
jalr	*	*	ALU	I	*	Reg	Imm	Add	Read	1	PC+4
jal	*	*	ALU	J	*	PC	Imm	Add	Read	1	PC+4
auipc	*	*	+4	U	*	PC	Imm	Add	Read	1	ALU

Caches

Index: set that memory will be placed in
Fully associative: no index bc only 1 set
N-way associative: #sets = #blocks/associativity
Direct mapped: #sets = #blocks
#bits_in_address = $\log_2$(#sets)
Tag: distinguish different blocks that have the same index
#bits_in_address= #address_bits_total - #index - #offset
Offset: location of the byte in block
#bits_in_address = $\log_2$(#bytes_in_block)
Address: TAG | INDEX | OFFSET
Cache capacity = associativity * #sets * block size in bytes
More associativity = more blocks per set
Index bits not stored with data, only tags and data itself are
Direct mapped: block can only go in one place within the cache
To find data, only need one compare, simpler!!! Just needs mux
# of sets = # of blocks
Equivalent to N-way associative where N = 1
N-way set associative: block can go in one of N places
To find data, have to do N compares
Total of (#blocks/N) sets
Fully associative: block can go anywhere in cache
To find data, have to do (# of blocks) compares, more complex, needs eviction strat as well. Must compare all tags
# of blocks = size of cache/size of block

Cache Misses

Compulsory – we haven't ever tried to pull this block of memory into the cache. Must miss – it's impossible that the data would be here
Solution: increase block size (increases miss penalty though)
Capacity – at some point in the past we had this data in our cache, but we had to evict it to make room for something else. Having a larger cache would help – if the cache were larger, this block would still be here
Solution: increase cache size (may increase access time)
Conflict – something else in our chain of accesses overwrote this block. Having a larger cache would not help because something else also maps to this block, so it would still be overwritten. It would help to have greater associativity.
Solutions: Increase cache size // Increase associativity (may increase access time) // Improve replacement policy (LRU etc)
Coherence – caused by coherence traffic with other processor, can dominate total misses in parallel programs [see Atomics]

Write policies:
There is a store instruction and the cache hits:
Write back = processor tells cache to update. Cache updates. Memory updates when data is evicted from cache back to memory (harder to implement, reduces # of writes) uses a dirty bit
Write-through = processor tells cache to update. Cache updates and tells memory to update (simpler, reliable)
There is a store instruction and the cache misses:
Write allocate = fetch data into cache after writing to memory, usually used with write back
No write allocate = only write to memory, usually used with write-through

Virtual Memory

Each process has own page table
log2(# PHYSICAL PAGES) == # bits for PPN
log2(# VIRTUAL PAGES) == # bits for VPN
log2(sizeof page) == # bits for offset (same for both VPN and PPN)
sizeof(memory [for either physical or virtual]) = #pages * sizeof(each page) == 2# address bits
entries in page table = # virtual pages
Valid → page is in physical memory – there can only be #physicalpages valid at a single time in Page Table!
Process:
Split the address into fields based on page size. Calculate the offset. The remaining bits are our VPN. check your TLB. Is there a valid mapping for your VPN?

- [TLB Hit] Convert to a physical address, update LRU/MRU bits in TLB
- [TLB Miss] check page table
  - [Page Table Hit] Convert to a physical address + update TLB by removing any entry (if fully-associative), or the LRU entry.
    - [Page Fault] create a new mapping. Check if memory is full. That is, if your physical memory space can hold four pages, are there four active mappings?
      - [No Eviction] Pick a new page on the 'free pages' list. Assign VPN and PPN mapping to both the page table and the TLB. Update LRU
      - [Eviction] Evict LRU mapping from the TLB and/or page table. Then, pick a new page on the 'free pages' list. Assign the VPN and PPN mapping to both the page table and the TLB. Update any LRU bits as necessary

1. Request data using the virtual address
2. Split virtual address into VPN, Offset fields
3. Access TLB with VPN
4. Access page table with VPN
5. Request new page from OS/Memory manager
6. Split physical address into PPN, Offset
7. Update page table with new PPN
8. Update TLB with new PPN, VPN
9. Adjust LRU bits on TLB
10. Return data

Caches (cont'd)

AMAT: Avg Mem Access Time
L1 Cache: AMAT = hit time + miss rate * miss penalty
Hit time not multiplied by hit rate bc cache is checked on both hits and misses
Larger caches: tradeoff between reducing miss rates and increasing hit time
Multilevel cache: AMAT = hit time L1 + miss rate L1 * (hit time L2 + miss rate L2 * miss penalty L2)
Global miss rate = (miss rate in largest cache) / (total # memory accesses)

Amdahl's Law: Parallelism is hard.

Speedup = $\frac{1}{(1-P) + \frac{P}{N}}$

where P is the proportion of code that can be sped up and N is the speedup ratio
Moral: Optimize the big parts of your code

RISC-V

Saved registers (sp, s0-s11) same before & after call
Volatile registers (t0-t6, a0-a7, ra) freely changed by callee
sw t0 [rs2], 4[imm](s0)[rs1]
Size of word: 4 bytes

Control Bit(s)	Question	Yes	No
BrEq, BrLt	Is this a branch inst?	0/1 (depends on if branch condition was satisfied)	*
PCSel	Are we jumping/branching?	ALUOut for jump, PC+4/ALUOut for branch (depends on if branch condition was satisfied)	PC+4
ImmSel	Is this an R inst?	*	I/S/SB/UJ/J
BrUn	Are we branching?	1 if unsigned, 0 otherwise	*
ASel	Are we jumping/branching?	PC	Reg
BSel	Is this an R inst?	Reg	Imm
MemRW	Are we storing in memory?	1	0
RegWEn	Are we modifying a register?	1	0
ALUSel	What math operation are we using? Check green sheet. (Will usually be "add", unless we are definitely not adding, like "mul".)		
WBSel	Is RegWEn 1?	PC+4 when we are storing PC+4; ALU output when we are storing the output of a math operation; Memory output when we are loading something from memory	*

inst	1	2	3	4	5	6	7	8	9	10	11	12	13	14
lw t0 0(a0)	IF	ID a0	EX	MEM r	WB t0									
beq t0,x0,end		NOP	NOP	IF	ID t0	EX	MEM	WB						
addi t0, t0, 2					NOP	NOP	IF	ID t0	EX	MEM	WB t0			
sw t0, 0(a0)								NOP	NOP	IF	ID t0 a0	EX	MEM w	WB

Caches (cont'd)

Increasing associativity ONLY → +hit rate, +hit time, =miss time
Increasing block size ONLY → =Hit rate, -hit time, +miss time
Increasing cache size by increasing the number of blocks (ex. Turning a direct-mapped cache into a 2-way set associative cache with doubled cache size)
→ +hit rate, +hit time, = miss time
Increasing cache size by increasing the number of sets
→ +hit rate, +hit time, =miss time
Increasing the size of main memory
→ =hit rate, =hit time, +miss time

Pipelining

t_clktoQ >= t_PC_clktoQ + t_IMEM_read + t_RFread + t_MUX + t_RFsetup
Fastest clock time = max(IF, ID, EX, MEM, WB)
For EX stage, branch comparator time usually overshadowed by ALU computation
Speedup from single cycle to pipelined less than 5 because

- Necessity of adding registers which have clk-to-q and setup times
- Need to set the clock to the maximum of five stages, which take diff amounts of times
- Dealing with hazards

Latency: Time for one instruction to go through the datapath // When changing from single instruction to pipeline, latency increases // Single cycle: add up delay for all steps // Pipelined: #stages * clock period
Throughput: Total instructions can be done in a certain time (second) // When changing from single instruction to pipeline, throughput increases //#stages / latency

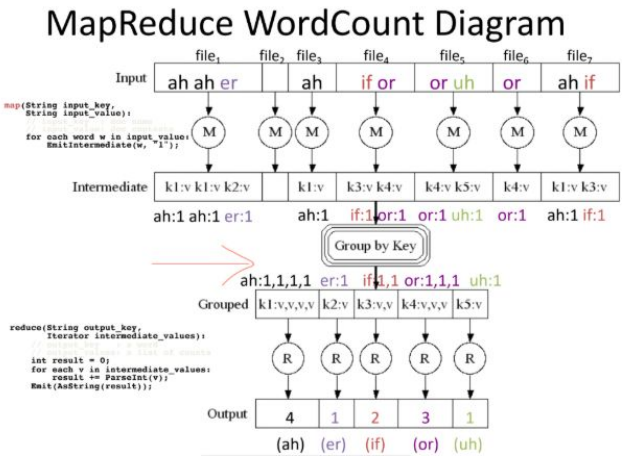
Hazards

Structural Hazards: When more than one instruction needs to use the same datapath resources at the same time
Causes:

1. Register File: accessed both during ID and WB. Solution: have separate read and write ports. To account for reads and writes to same register: processors write to register during first half of clock cycle, read during second half. AKA DOUBLE PUMPING
2. Memory: accessed for both instructions and data. Solution: have separate IMEM and DMEM

In general – structural hazards can always be resolved with more hardware

Data Hazards: Caused by data dependencies; when instruction **reads** from register before a previous instruction has finished **writing** to that register
Solution:
Forwarding: when result of the EX or MEM stage sent to EX stage for following instruction to use – see discussions for examples
Stalling: use a nop to stall the operation
Control Hazards: Caused by jump and branch instructions
Solution: stalling? Slow. Use predict and kill!
STAGES OF DATAPATH
Fetch: Send address to instruction memory, read IMEM at that address
Decode: Generate control signals from the instruction bits, generate the immediate, and read registers from the RegFile, branch prediction also happens here
Execute: Perform ALU operations and do branch comparison
Memory: Read from or write to data memory
Writeback: Write back either PC + 4, the result of the ALU operation, or data from memory to the RegFile



MapReduce and Spark

Resilient Distributed Datasets (RDD): Primary abstraction of distributed collection of items
Transforms (RDD → RDD)
Map (f): Returns a new transformed item formed by calling f on a source element
flatMap(f): Similar to map, but each input item can be mapped to 0 or more output items, return a sequence of outputs.
reduceByKey(f): returns dataset of (K,V) pairs where values for each key aggregated using reduce function
Actions (RDD → Value)
Reduce(f): aggregate elements of dataset regardless of keys using function f
Map: for each element in a long list, run function to convert it to different form. Gives one element to each thread, runs in O(1) time
Reduce: takes in two elements of a list, outputs another element of the same type. Randomly reduce two elements at t a time until left with only one element, reduce in tourney bracket style, runs in O(log n) time.
Overhead: takes time to send info to many computers
Requirements
Map: must be run on only one element of a list: Output multiple outputs, as long as of same type.
Reduce: Want reduces to run in O(1) time, be commutative and associative
No point of having multiple maps or reduces in a row

Atomics & Cache Coherency

Synchronization: limits access to shared resource to 1 actor at a time – only 1 person permitted to edit a file at a time, otherwise changes by several people get all mixed up
Solution: Take turns!
Locks:
control access to shared resources
Like a microphone shared between people, aka “semaphore”
Implemented with a variable, 0 for unlocked, 1 for locked
Lock Synchronization
What if we have different threads? Both threads will think they have the lock and access to edit, but this is not valid!
SOLUTION: Hardware Synchronization, Atomic read/write
Read and write in single instruction, no other access permitted between read and write
Of course, overhead
Must use shared memory!
Common implementations:
Swap register with memory ← focus on this for simplicity in this course!
Pair of instructions for “linked” read and write
Write fails if memory location has been “tampered” with after linked read
RISC-V Atomic Memory Operations (AMOs)
Atomically perform an operation on an operand in memory and set destination register to og memory value
Use R-type, but with extension
Example: RISC-V Critical Section
Assume lock is in memory location stored at a0
Lock “set” at 1, “free” at 0 [initial]

```
# lock at a0
li t0 1                                  # Get 1 to set lock
Try: amoswap.w.aq t1, t0, (a0)           # t1 gets old lock value
bnez t1, Try                             # if it was already 1, try another

----------CRITICAL SECTION, DO YO STUFF----------
amoswap.w.rl                             # store 0 in lock to release
```

Amoswap.w.aq = reads from mem onto register, swaps info // w = word // aq = acquire
Amoswap.w.rl = rl = release

Comparison of Methods:

Lock Synchronization

Broken Synchronization

```
while (lock != 0) ;

lock = 1;

// critical section

lock = 0;
```

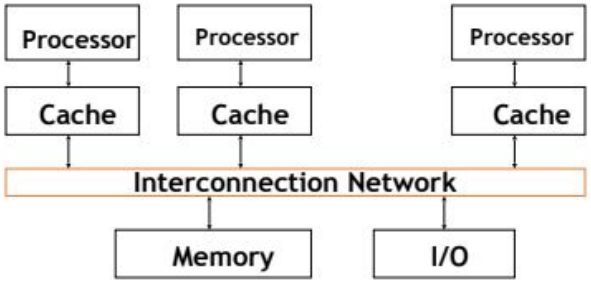
Fix (lock is at location (a0))

```
        li          t0, 1
Try:    amoswap.w.aq    t1, t0, (a0)
        bnez        t1, Try
Locked:

        # critical section

Unlock:
        amoswap.w.rl    x0, x0, (a0)
```

Synchronization in OpenMP
Typically are used in higher level parallel pgming constructs
Critical section: only one thread at a time can enter a critical region (threads wait their turn
Deadlock: system state when no progress is possible
Solution?
Double omp_get_wtime(void); returns wall clock time in seconds
Time measured per thread, no guarantee that two distinct threads measure the same time
Measured from “some time in the past” so subtract results of two calls to omp_get_wtime to get elapsed time
Multicore Multiprocessor:

Atomics & Cache Coherency (cont'd)

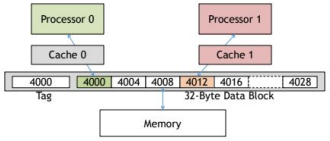
SMP: Shared Memory Symmetric Multiprocessor
Two or more identical CPUs/cores
Single shared coherent memory
Single address space shared by all processors/cores
Processors coordinate/communicate through shared variables in memory (via loads and stores)
Use of shared data must be coordinated via synchronization primitives(locks) that allow access to data to only one processor at a time
All multicore computers today are SMP
Multiprocessor Caches
Memory – performance bottleneck
Use caches to reduce bandwidth demands on main memory
Each core has local private cache holding data accessed recently
Only cache misses have to access the shared common memory
Keeping Multiple Caches Coherent



Architect's job: shared memory → keep cache values coherent
IDEA: When any processor has cache miss or writes, notify other processors via interconnection network
If only reading, many processors can have copies
If processor writes, invalidate any other copies
Write transactions from one processor, other caches "snoop" the common interconnect checking for tags they hold
Invalidate any copies of same address modified in other cache
How does Hardware Keep Cache Coherent? Each cache tracks the state of each block in cache
Shared: up to date data, other caches may have a copy
Modified: up to date data, changed (dirty), no other cache has a copy == ok to write, write back later
Exclusive: up to date data, no other cache has a copy, Ok to write, memory up to date
Avoids writing to memory if block replaced
Supplies data on read instead of going to memory
Owner: up to date data, other caches may have a copy (they must be in shared state)
Cache is one of several with valid branch of cache line, but has exclusive right to make changes (like master branch vs other branches). Must broadcast changes to all other caches sharing the line (git pull!). Allows dirty sharing of modified data without updating main memory. Cache line may be changed to Modified, or changed to Shared state (set branch as master, git push, respectively). Owned cache lines must respond to snoop request with data.

False sharing: block ping-pongs between two caches even though processors are accessing disjoint variables.
In the example below, processor 0 and processor 1 are accessing two different elements… but because these two elements are in the same block, and cache coherency is managed at the block level, the block (and thus the data for both elements) will be invalidated each time either processor writes to their respective element. It is called "false sharing" because from the hardware's perspective it seems like the processors are sharing the same data, but really they are not— they are accessing different data in the same block.

Cache Coherency Tracked by Block



- Suppose block size is 32 bytes
- Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
- Suppose in X location 4000, Y in 4012
- What will happen?

Sanity check!

I. If in Exclusive state, processor can write without notifying other caches
II. Owner state is variation of Shared state to let caches supply data instead of going to memory on read miss
III. Exclusive state is variation of Modified state to let caches avoid writing to memory on a miss

ALL TRUE!!!

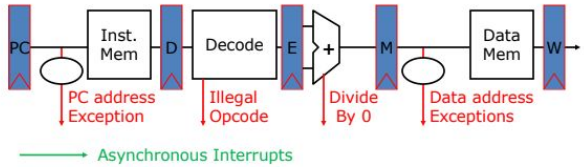
I/O

Processor needs:
Input: read a sequence of bytes
Output: write a sequence of bytes
Interface options:
Special I/O instructions and hardware
Memory mapped I/O
Portion of address space dedicated to I/O
Only I/O registers, no memory
Use normal load/store instructions
V common! Used by RISC-V
Polling:
Device registers → two functions
Control register (ok to read/write)
Data register (contains data)
Processor reads from Control Register in loop
Wait for device to set ready bit in control register (0→1)
Indicates "ready to accept data" // "data available"
Processor loads from or writes to data register
Cost of polling: acceptable for mouse (low polling rate), unacceptable for hard disk (high polling rate)
Solution?
Interrupt: caused by an event external to current running program (key press, disk I/O) – asynchronous to current program; can handle interrupt on any convenient instruction
Exception: caused by some event during execution of one instruction of current running program (ie divide by 0, bus error, illegal instruction etc)

Trap: action of servicing interrupt or exception by hardware jump

Precise Traps
Every instruction prior to the trapped one has completed, no instruction after the trap has executed
Implies that handler can return from an interrupt by restoring user registers and jumping back to interrupted instruction
Interrupt handler software doesn't need to understand the pipeline of the machine, or what program was doing
Exception MORE COMPLEX than interrupt
Providing precise traps tricky, but necessary for virtual memory & other things to function properly

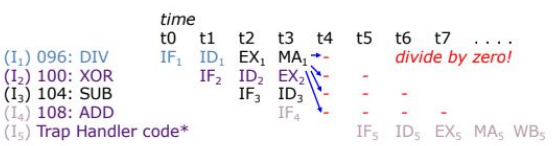
Trap Handling in 5-Stage Pipeline



Exceptions are handled *like pipeline hazards*

1) Complete execution of instructions before exception occurred
2) Flush instructions currently in pipeline (i.e., convert to **nop**s or "bubbles")
3) Optionally store exception cause in status register
   - Indicate type of exception
   - Note: several exceptions can occur in a single clock cycle!
4) Transfer execution to trap handler

Trap Pipeline Diagram



*MEPC = 100 (instruction following offending address, here DIV)

Programmed I/O :
CPU execs lw/sw instructions for all data movement to/from devices
CPU spends time doing two things:
Getting data from device to main memory
Using data to compute

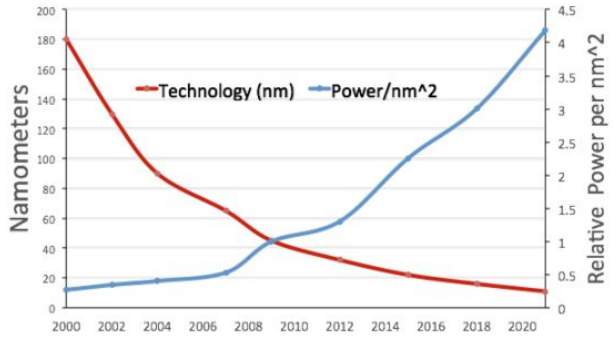
Cloud Computing & WSC

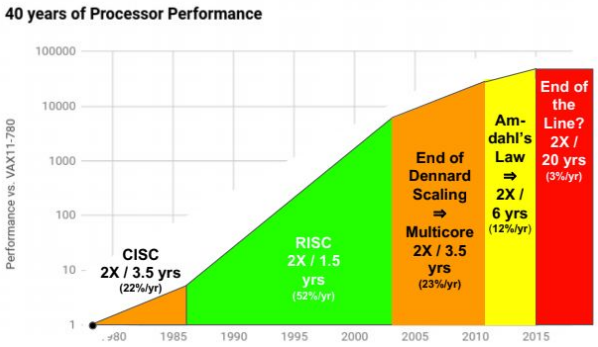
"The Web Space Race" – build-out driven by growth in demand (more users)
Economy of scale! 5-7x cheaper
More pervasive broadband Internet
Commoditization of HW & SW
Massive scale datacenters: 10K to 100K servers + networks to connect them together
Emphasize cost-efficiency, power important because WOW big
Relatively homogeneous HW/SW

Very large applications (internet stuff mainly to support the 5 websites that we're on every day): reddit, twitter, facebook, gmail, youtube
Cope with failures common at scale
Design Goals?
Ample parallelism:
Batch apps: large number independent data sets with independent processing
Operational Costs Count:
Cost of equipment purchases << cost of ownership
Equipment: Server, Rack, Array
Latency/response time: time between start and completion of a task
Throughput/bandwidth: total amount of work in a given time
Takeaway: WSC has to be super efficient
Goal of WSC: % peak load = % peak energy
PUE: (computing + building power, basically total power) / computing equipment power
Power efficiency measure for WSC, not including efficiency of servers, networking gear
1.0 = perfecto

Golden Age of Computer Architecture

Software advances can inspire architecture innovations
Raising HW/SW interface creates opportunities for architecture innovation
Marketplace settles architecture debates
Processor designs split between datapath and control
As you know from CPU project, control is hardest!
Microprogramming: logic used to design control
We are now in post Moore's Law Era
Dennard Scaling: energy scaling for fixed task is better, since more and faster transistors

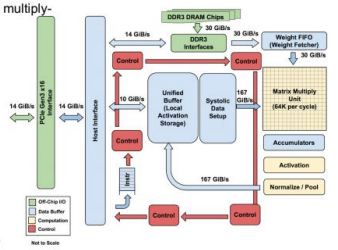




End of Growth of Single Program Speed?
Current Security Challenge
Speculation ⇒ timing attacks
Microarchitecture attacks
Spectre is but in computer architecture
Software not yet secure, how can hardware help?
What Opportunities Left?
SW-centric
Modern scripting languages interpreted, dynamically-typed and encourage reuse – efficient for programmers but not for execution
HW-centric
Only path left – Domain Specific Architectures; do a few tasks, but extremely well
Achieve higher efficiency by tailoring architecture to characteristics of domain
More effective parallelism
More effective use of memory bandwidth
Eliminate unneeded accuracy
Provides path for software
Raise level of HW/SW interface

TPU: High-level Chip Architecture

- The Matrix Unit: 65,536 (256x256) 8-bit multiply-accumulate units
- 700 MHz clock rate
- Peak: 92T operations/second
  - 65,536 * 2 * 700M
- >25X as many MACs vs GPU
- >100X as many MACs vs CPU
- 4 MiB of on-chip Accumulator memory + 24 MiB of on-chip Unified Buffer (activation memory)
- 3.5X as much on-chip memory vs GPU
- 8 GiB of off-chip weight DRAM memory



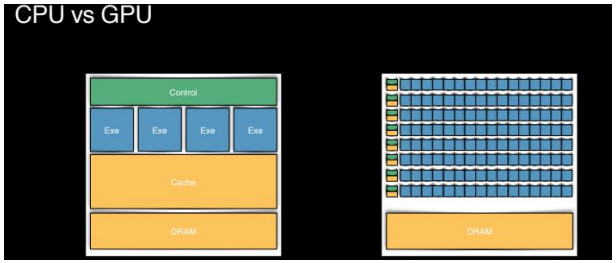
25

GPU

Specialized co-processor to render 3D objects and simulate light effects
Parallel processor leveraged for general compute, including machine learning
Integrated HW/SW system
Challenges? Complex and evolving HW/SW features // Rapid evolution // Scaling, determining image correctness

	CPU	GPU
Cores	Fewer	Many
Frequency	Higher	Lower
Latency	Low	Higher
Parallelism	Instruction level	Thread level
Registers	Fewer (Register renaming	Many
Speculation	Highly leveraged, br prediction	Limited (no br prediction)
Execution Order	Out of order (reorder buffers	In order
Execution Units	Fewer	Many
Execution ctrl	Complex	Simpler
Coherency	Hardware Managed	Software manged

CPU vs. GPU
CPU: minimize latency of limited threads using complex control
GPU: maximize throughput by scheduling many parallel threads



Shaders
Programs that execute on the GPU
Written in a C-like shader language
Compiled, linked, and combined into a larger program
Vertex Processing
Primitives: points, lines, triangles
Vertex shading: determine position, transformations, evaluate attributes
Transformation and projection: rotation, translation, and scaling of geometry. distant objects are smaller
Clipping and culling: remove offscreen and hidden geometry for efficiency
Rasterization
Triangle Setup/Primitive Assembly: differentials, edge equations
Triangle traversal: iterate and check each sample for coverage, generate fragments
Anti-aliasing: multiple techniques with different tradeoffs
Fragment Processing
Interpolation of fragments: simplest determination based on vertex attributes
Per pixel shading computations: creates more complex effects than simple interpolation
Texture mapping: mapping an image onto a triangle
Lighting: how surfaces respond to light and project colors
Per fragment tests: check each pixel location

```
void reverse(struct node * head) {
   struct node * prev = NULL;
   struct node * next;
  struct node * curr = head;
  while(curr != NULL){
    next = curr->next;
    curr->next = prev;
    prev = curr;
    curr = next;
      }
  }
```

1. reverse: addi sp sp -12
2. sw s0 0(sp)
3. sw s1 4(sp)
1. sw s2 8(sp)
2. add s0 a0 x0
3. xor s2 s2 s2
4. loop: beq s0 x0 exit
5. lw s1 4(s0)
6. sw s2 4(s0)
7. add s2 s0 x0
8. add s0 s1 x0
9. j loop
10. exit: lw s0 0(sp)
11. lw s1 4(sp)
12. lw s2 8(sp)
13. addi sp sp 12
14. j ra

Jump Instructions

jal is jump-and-link, What this instruction does is that it stores PC+4 into the register destination and jumps to a certain offset from the given PC (jal rd offset)

ex: (jal ra 20) This will STORE PC+4 into ra and jump to PC+20. This is used when you only know the offset of where you want to go and also want to remember where you are.

Side note 1: So Jal can also be used with a label instead of an offset. Using jal ra Square, this will STORE PC+4 into ra and jump to the Square label. This is technically a pseudo-instruction.

Side note 2: If there is no specified rd (jal Label), then pretend there is an implicit register “ra”. It acts as if it’s also a pseudo-instruction. So (jal Square) is really jal ra Square.

jalr is jump-and-link-register, this instruction jumps to a specific address stored in rs1 plus offset, and also stored PC+4 into rd. (jalr rd rs1 offset)

assuming rs1 contains the address of some label Square

ex: (jalr ra rs1 0) We will also store current PC+4 into register ra. Then we will jump to where rs1 is plus offset of 0

, so it's just where Square is located in memory. Jumping means PC will be set to that location.

This is used when we have the address of the location we want to go to, also might include an offset, and it lastly remembers where you are now.

j is jump, this is a pseudo-instruction for jal that jumped to a certain label without storing where PC+4 is.(j Label)

KEY POINT: j Label IS THE SAME AS jal x0 Label.

ex: (j Loop) This basically jumps PC to be where Loop is located. That's it~ It "implicitly" stores PC+4 into x0, but this actually does nothing since editing x0 does nothing.

We want to use j when we only have the label of where to go, AND don't want to remember where you are.

jr is Jump-Register, this is ALSO a pseudo-instruction for jalr. It will jump to a certain address stored in rs1 with NO offset, and doesn't store PC+4 into anywhere(jr rs1)

KEY POINT: jr rs1 IS THE SAME AS jalr x0 rs1 0

Assuming ra contains the address of the label Loop

ex: (jr ra) We store PC+4 into x0, which does nothing because x0 can't be edited. This instruction jumps to the location that ra contains with NO offset.

We want to use jr when we want to jump to a memory address stored in a register(and no offset) and we don't want to remember where we are.

j/jal relative addressing, jr/jalr absolute addressing

```
Solution: def parseLine(line):
    tokens = line.split(" ")
    types = tokens[2:]
    results = []
    for type in types:
        results.append((type, (tokens[0], 1))
    return results


def reduceFunc(v1, v2):

    return (v1[0] + v2[0], v1[1] + v2[1])


def average(k, v):

    return (k, v[0] / v[1])


pokemonData = sc.parallelize(pokemon)

out = pokemonData.flatMap(parseLine)

                 .reduceByKey(reduceFunc)

                 .map(average)
```