Cheat sheet
by Kenny Luu, page 1 of 2

1 CPU Power Consumption

Chip Power Consumption $= C * Vdd^2 * F$ C = capacitance, Vdd = voltage, F = frequency

Dennard scaling: Transistors get smaller but their power density stays the same. The supply voltage of a chip can be reduced by $0.7x$ every generation

$Power = C * Vdd^2 * F_{0\to 1} + Vdd * I_{leakage}$ Leakage gets worse with samller devices and lower Vdd, it also gets worse with higher temps

Amdahl's Law: We want to make the common case efficient, given an optimization x that accelerates fraction f_x of the program by a factor S_x, the overall speedup is: $\frac{1}{(1-f_x)+\frac{f_x}{S_x}}$

Performance

Latency: how long it takes to do a task

Throughput: total work done per unit time

$ExeTime = \frac{Instrs}{Program}\frac{Clockcycles}{Instr}\frac{Sec}{ClockCycle}$ **Relative Performance:** define performance as $= 1/ExecutionTime$ "X is n times faster than Y" means $Perormance_x/Performance_y$

2 Instruction Set Architechture

CISC	RISC
Emphasis on Hardware	Emphasis on Software
Multi-cycle complex instructions	Simple (single clock) instructions
Memory-to-Memory load/store incorporated in instr.	Register-to-Register Separate load/store instructions
Small code size	Large code size
High CPI	Low CPI
Low clock frequency	High clock frequency
Variable length instructions	Same length instructions
Complex instruction decode	Simple instruction decode
HW difficult to implement	HW easy to implement

R-type

funct7	rs2	rs1	funct3	rd	opcode

Opcode: basic operation of instruction (7) Rs1: Register source 1 operand (5)
Rd: Register destination operand (5) Rs2: register source 2 operand (5)
Funct3: additional opcode field (3) Funct7: additional opcode field (7)

Question: Why did RISC-V only define 32 registers?

I-type Question: What is an immediate?

immediate[11:0]	rs1	funct3	rd	opcode

S/B-type Question: Why is the immediate split?

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

U/J-type

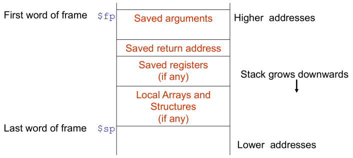
immediate[31:12]	rd	opcode

Constant == immediate == literal == offset

There are 32 registers in RISC-V **ld dst, offset(base)** the base is the starting address of the array the offset is the index

When using switch statement we can use a jump table, a jump table holds addresses in memory of where the code for the jump targets are.

Stack: The stack is a allocated in frames, it stores the state of a procedure for a limited time, the callee returns before the caller does. The things which can be saved on the stack are: local arrays, return addresses, saved registers, and nested call arguments



Callee Saved	Caller Saved
Saved registers ($s0-$s7)	Temporary registers ($t0-$t9)
Stack/frame pointer ($sp, $fp, $gp)	Argument registers ($a0-$a3)
Return address ($ra)	Return values ($v0-$v1)

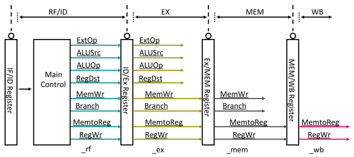
- Callee saved registers (preserved for caller)
  - Save register values on stack prior to use
  - Restore registers before return
- Caller saved registers (not preserved for caller)
  - Do what you please and expect callees to do likewise
  - Should be saved by the caller if needed after procedure call

Procedure Call Steps

1. Place parameters in a place where the procedure can access them
2. Transfer control to the procedure
3. Allocate the memory resources needed for the procedure
4. Perform the desired task
5. Place the result value in a place where the calling program can access it
6. Free the memory allocated in (3)
7. Return control to the point of origin

3 Pipelining

In pipelinging we overlap instructions in defferent stages



There are hazards, these include **structural hazards** where a required resource is busy, **data harzards** where we must wait previous instructions to produce/consume data, and **control hazards** where next PC depends on previous instruction.

Structural Hazards

two instructions are trying to use the same hardware within the same cycle, to solve this we can make all the instructions the same length

Data Dependencies

Dependencies for instruction j following instruction i

- Read after Write (RAW or true dependence)

  Instruction j tries to read before instruction i tries to write it
- Write after Write (WAW or output dependence)

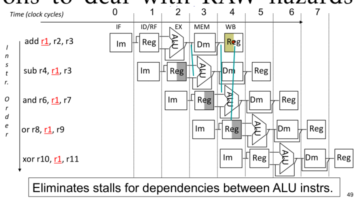
  Instruction j tries to write an operand before i writes its value
- Write after Read (WAR or (anti dependence))

  Instruction j tries to write a destination before it is read by i

Solutions for RAW Hazards: We can delay the reading of an instruction until data is available, to do this we can insert pipeline bubbles, can also write to the register file in the first half of a cycle and then read in the second half.

Forwarding: Another solution is forwarding or pushing the data to an appropriate unit.

We can also reorder instructions to deal with RAW hazards



- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

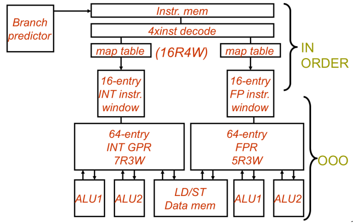
Control Hazards

A control hazards is like a data hazard on the PC, we cannot fetch the next instruction in the pipeline if we don't know the PC

Some solutions for control hazards are stalling on branches, predicting taken or not taken. We need to flush the pipeline if we predict wrong, in a 5-sage pipeline we only need to flush 1 instruction

4 Out of Order Execution and ILP

We want to avoid in-order stalls so we use out of order execution to re-order instructions based on dependencies



A **superscalar processor** is a CPU that implements a form of parallelism called instruction-level parallelism within a single processor. I.e we can launch multiple instructions every cycle.

There are some issues with multiple instructions executing at onc,e we need to double the amount of hardware, we introduce hazards, branch delay, & load delay

We can rename (map) architectural registers to physical registers in decode stage to get rid of false dependencies

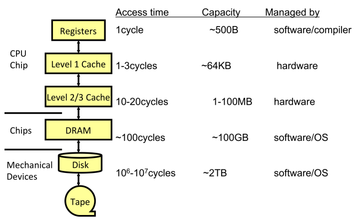
Superscalar + Dynamic scheduling + register renaming

```
add  $t0A, $t1, $t2     sub $t0B, $t1, $t2
or   $t3, $t0A, $t2     and $t5, $t0B, $t2
```

There are some limits to **ILP** and pipelining:

- Limited ILP in real programs
- Pipeline overhead
- Branch and load delays exacerbated
- Clock cycle timing limits
- Limited branch prediction accuracy (85%-98%)
- Even a few percent really hurts with long/wide pipes
- Memory inefficiency
- Load delays + # of loads/cycle

5 Caches



Principle of locality Programs work on a relatively small portion of data at any time, so we can predict data accessed in near future by looking at recent accesses

There are two types of locality: **spatial** and **temporal**, **spatial** locality is if an item has been accessed recently, nearby items will tend to be referenced soon, and **temporal** is if an item has been referenced recently, it will probably be accessed again soon

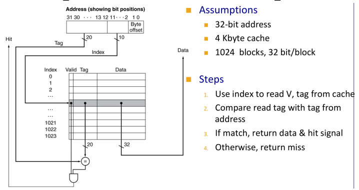
How is data stored in a cache? There are three ways:

- Direct mapped (single location)
- Fully associative (anywhere)
- Set associative (anywhere in a set)

Direct Mapped Cache

Location in cache determined by (main) memory address, we use the lowest order bits to determine this address.

To determine which particular block is stored in a cache location, we look at **tag** and **valid** bits, the **tag** bits are the high-order bits of the memory address, the **valid** bit represents 1 = present, 0 = not present



- *Block* – Minimum unit of data that is present at any level of the hierarchy
- *Hit* – Data found in the cache
- *Hit rate* – Percent of accesses that hit
- *Hit time* – Time to access on a hit
- *Miss* – Data not found in the cache
- *Miss rate* – Percent of misses (1 - Hit rate)
- *Miss penalty* – Overhead in getting data from a higher numbered level
  - Miss penalty = higher level access time + Time to deliver to lower level + Cache replacement / forward to processor time
  - Miss penalty is usually much larger than the hit time
  - This is in addition to the hit time
- These apply to each level of a multi-level cache
  - e.g., we may miss in the L1 cache and then hit in the L2

Average Memory Access Times:

$Accesstime = hittime + missrate \times misspenalty$

3 C's of Cache Misses:

- Compulsory – this is the first time you referenced this item
- Capacity – not enough room in the cache to hold items

  this miss would disappear if the cache were big enough
- Conflict – item was replaced because of a conflict in its set

  this miss would disappear with more associativity

$CPIpenalty = missrate \times misspenalty$

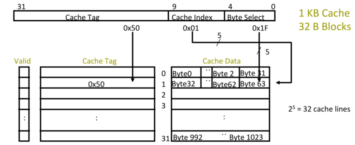
When we encounter a miss the pipeline stalls for an instruction or data miss

Cache Blocks

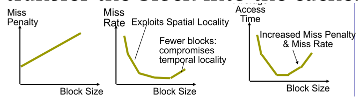
Assuming a 2^nbyte direct mapped cache with 2^m byte blocks

- Byte select – The lower m bits
- Cache index - The lower (n-m) bits of the memory address
- Cache tag - The upper (32-n) bits of the memory address

Direct Mapped Problems: **Thrashing** If accesses alternate, one block will replace the other before reuse



Block Sizes: Larger block sizes take advantage of spatial locality, however it also incurs larger miss penalty since it takes longer to transfer the block into the cache.

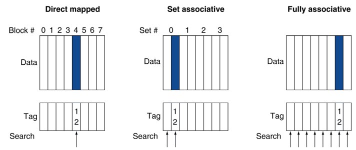


Fully Associative Cache

Opposite extreme in that it has no cache index to hash, it uses any available entry to store memory elements, there are no conflict misses, only capacity misses, and we must compare cache tags of all entries to find the desired one

N-way Set Associative Cache

Compromise between direct-mapped and fully associative, each memory block can go to one of N entries in cache, and each set can store N blocks; a cache contains some number of sets



Tag & Index with Set-Associative Caches

Given a 2^n byte cache with 2^m byte blocks that is 2^a set-associative, the cache contains 2^{n-m} blocks, and each cache way contains 2^{n-m-a} blocks, and the

Cheat sheet
by Kenny Luu, page 2 of 2

cache index is $n - m - a$ bits after the byte select

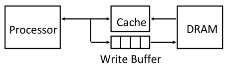
Organization	# of sets	# blocks / set	12 bit Address
Direct mapped	16	1	tag 4, index 4, blk off 4
2-way set associative	8	2	tag 5, index 3, blk off 4
4-way set associative	4	4	tag 6, ind 2, blk off 4
8-way set associative	2	8	tag 7, i 1, blk off 4
16-way (fully) set associative	1	16	tag 8, blk off 4

Some cons with associative caches are that there is an an area overhead and more latency

The **replacement policy** for a N-way set associative cacheis choosing the least recently used thing

Cache Write Policies

- Write-through (write data go to cache and memory)
  - Main memory is updated on each cache write
  - Replacing a cache entry is simple (just overwrite new block)
  - Memory write causes significant delay if pipeline must stall
- Write-back (write data only goes to the cache)
  - Only the cache entry is updated on each cache write so main memory and the cache data are inconsistent
  - Add "dirty" bit to the cache entry to indicate whether the data in the cache entry must be committed to memory
  - Replacing a cache entry requires writing the data back to memory before replacing the entry if it is "dirty"



Use Write Buffer between cache and memory
- Processor writes data into the cache and the write buffer
- Memory controller slowly "drains" buffer to memory

Write Buffer: a first-in-first-out buffer (FIFO)
- Typically holds a small number of writes
- Can absorb small bursts as long as the long-term rate of writing to the buffer does not exceed the maximum rate of writing to DRAM

Write Miss Options:

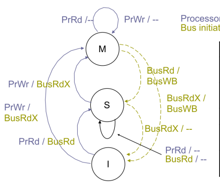
Steps	Write through, Write allocate: fetch on miss	Write through, Write allocate: no fetch on miss	Write through, No write allocate: write around	Write through, No write allocate: write invalidate	Write back, Write allocate: fetch on miss	Write back, Write allocate: no fetch on miss
1	pick replacement	pick replacement			pick re-placement	pick re-placement
2				invalidate tag	[write back]	[write back]
3	fetch block				fetch block	
4	write cache	write partial cache			write cache	write partial cache
5	write memory	write memory	write memory	write memory		

Splitting Caches Most chips have separate caches for instructions and data, some advantages are that we have extra bandwidth and low hit time, but miss rate will be higher.

Multilevel Caches: Different levels of caches L1, L2, and L3. L1 is focused on hit time L2 is focused on hit rate, L3 is extra.

Cache Coherence: State and sequence of actions needed to ensure caches are coherence in multicore systems, there are protocols that rely on monitoring other caches.

MSI Coherence Protocol



Abbreviation	Action
PrRd	Processor Read
PrWr	Processor Write
BusRd	Bus Read
BusRdX	Bus Read Exclusive
BusWB	Bus Writeback

6 DRAM

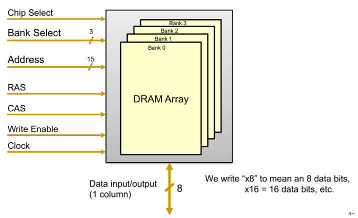
Memory-Access Protocol

5 basic commands
- ACTIVATE (open a row)
- READ (read a column)
- WRITE
- PRECHARGE (close row)
- REFRESH

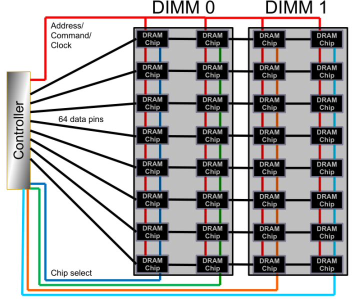
To reduce pin count, row and column share same address pins
- RAS = Row Address Strobe
- CAS = Column Address Strobe

Modern DRAM chips consist of multiple banks, they operate independently, but share command/address/data pins, each bank can have a different row active so we can overlabp ACTIVATE and PRECHARGE latencies, e.g. READ to bank 0 while ACTIVATING bank 1

DRAM High Level



Ranks A 64-bit DIMM with 16 chips with x8 interfaces has 2 ranks, each 64-bit group of chips is called a rank, all chips in a rank respond to the same command, different ranks share command/address/data lines, select between different ranks whit chip select signal. Ranks provide more "banksäcross multiple chips



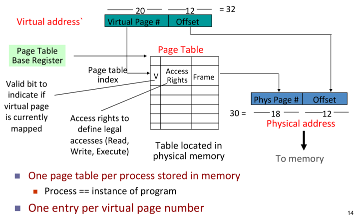
DRAM Channels All DIMMs get the same command, one of the ranks replies

Multi-Channel: Multiple lockstep channels, single channel with wider interface which gives faster cache line refill.

7 Virtual Memory

We expose a virtual memory space to applications so that they can assume that they own the whole space, then the OS handles the mapping of virtual to physical memory space. Every program will have its own virtual address space that is protected from other programs. Frequently-used portions of virtual address space copied to DRAM.

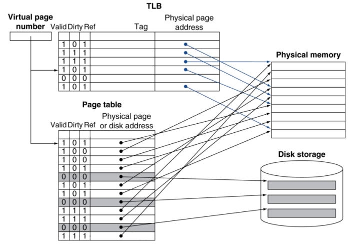
Page Table: A data structure which contains virtual to physical address mappings. If we haven't used a page in a while we can map it to disk memory instead of DRAM. Page tables are mapping in a coarser granularity i.e 4Kb



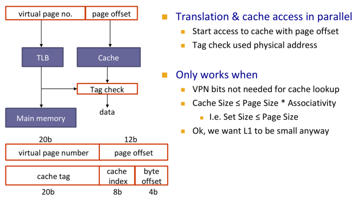
- One page table per process stored in memory
  - Process == instance of program
- One entry per virtual page number

TLB is a hardware cache just for translation entries for page tables, locality in accesses → locality in translations. Each TLB entry stores a page table entry (PTE), the data that is stored is **physical page numbers, permission bits (RXW), other PTE info (dirty bit, etc.)**

TLB Organization



Virtually Indexed, Physically Tagged Cache



8 I/O

Throughput-Latency Tradeoff:
$T = T_0 \frac{\rho}{1-\rho}$

Hard Disks Each sector of a hard disk records Sector ID and data, ECC, synchronization feilds and gaps, to access a sector you need to: **seek(move the head), rotational latency, data transfer, controller overhead**

Disk performance: seek time + $\frac{.5}{RPM/60} + \frac{sector}{TR}$ + controller delay

HW/SW IO interface Two common approaches for this are using registers and buffers or **Memory mapped IO**, in registers there are data registers and command registers. In memory mapped IO portions of physical addresses are assigned to each IO device. IO addresses correspond to device registers and buffers. IO address space is protected so user programs cannot access it

Physical IO interface Common interfaces include **busses** and **point-to-point links & switch-based interconnects**

Bus A shared communication link that connects multiple IO devices, it is a single set of wires which connect in parallel e.g. 32-bit bus = 32 wires of data.

Bus Transactions

An initiator acquires the bus
- May require arbitration

Initiator starts a transaction
- E.g., the CPU reads or writes a word to memory
- Specifies command (read, write, reset, etc...)
- Specifies address (which memory, device/register to access)
- Specifies outbound data for a write

Target responds to the transaction
- Does the specified action (e.g., reads or writes the data)
- Responds (possibly with data – for a read)

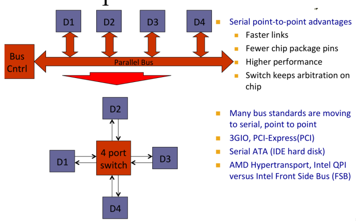
Synchronous Bus

Synchronous Bus
- Includes a clock in control lines
- Fixed protocol for communication relative to the clock
- Advantage: involves very little logic and can therefore run very fast
- Disadvantages
  - Every decision on the bus must run at the same clock rate
  - To avoid clock skew, bus cannot be long if it is fast
- Example: processor-memory Bus

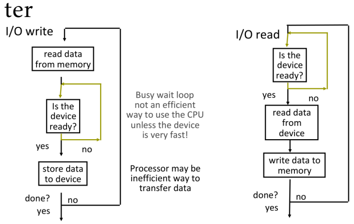
Asynchronous Bus
- No clock control line
- Can easily accommodate a wide range of devices
- No clock skew problems, so bus can be quite long
- Requires handshaking protocol

Increasing bus bandwidth To increase bandwidth we can transfer in blocks, transferring a burst of data. We can also pipeline the bus, we increase the clk frequency and have mulitple lines of data in flight at once. The advantage of a bus is that it's simple and has broadcast and is serialized, but it's slow

Point-to-Point Links & Switches Allows to have multiple transactions going on at once, an example is PCI-e. Also allows much higher clock frequencies.



IO notification w/ polling OS periodically checks the status register



IO notification w/ interrupts Just like an exception but due to en external source, execution is only halted during actual transfer.

Direct Memory Access (DMA) A custom engine for data movement, transfer blocks of data to or from memory without CPU intervention. The processor sets up DMA by giving the indentity of the device and the memory address for source/dest DMA engines are arbiters.

9 Custom Hardware

Instruction power efficiency Many instructions have a large overhead, the an entire instruction can take up to 70pJ, but the operation might take 0.5pJ, meaning that the overhead is 99%. To solve this we can amortize instructions by performing a large number of operations per instruction, and for each data cache access

Data-Level Parallelism We can use **vector** instructions to do a single operation on many values. To do this we need to add vector registers, as set a small values.

```
vadd.vv v3, v1, v2
```

SIMD single instruction muliple data instructions.

FPGA's are Field-programmable Gate Arrays, they are 2-D arrays of reconfigurable logic elements. They are used for digital circuit emulation and as custom hardware accelerators

Multicore In a multicore system we encounter the problem of synchronizing data, to fix this we can lock parts of memory when we are modifying it, and then unlock it when we are done.

Multithreading When we are multithreading we execute multiple threads on a single core, threads within a process will have separate PC and registers, but will share the same virtual memory space. When 1 thread stalls, switch to another thread.