Cheat sheet
by Kenny Luu, page 1 of 2

1 CPU Power Efficiency & Consumption

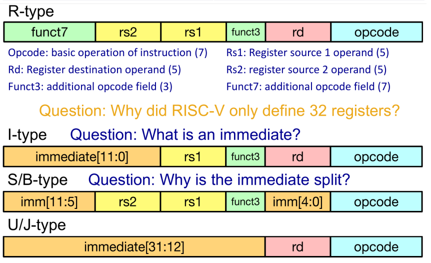
Chip Power Consumtion $= C * Vdd^2 * F$
$C =$ capacitance, $Vdd =$ voltage, $F =$ frequency
Dennard scaling: As transistors get smaller their power density stays contant, The supply voltage of a chip can be reduced by $0.7x$ every generation
Dennard scaling held up in the past but now it's not holding up
$Power = C * Vdd^2 * F_{0 \to 1} + Vdd * I_{leakage}$
Leakage gets worse with samller devices and lower Vdd, it also gets worse with higher temps
$Energy = AvgPower * ExecutionTime$
$Joules = Watts * sec$
Amdahl's Law-
We want to make the common case efficient, given an optimization x that accerlerates 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

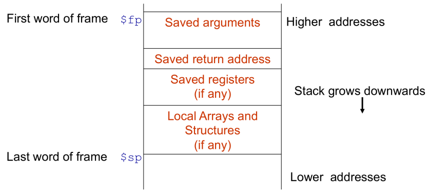
CISC	RISC
Emphasis on Hardware	Emphasis on Software
Multi-cycle complex instructions	Simple (single clock) instructions
Memory-to-Memory load/store incorporated in instr.	Register-to-Register Separate load/store instructions
Small code size	Large code size
High CPI	Low CPI
Low clock frequency	High clock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



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?
Question: What is an immediate?
Question: Why is the immediate split?

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.
Procedures: are required for structured programming, to implement them in assembly you need a memory space for local vars, arguments must be passed in and return values must be passed out, execution must continue after the call.
Stack: The stack is a LIFO data-structure 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)

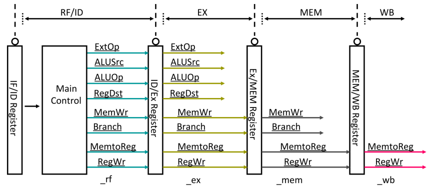
- Callee saved registers (preserved for caller)
  - Save register values on stack prior to use
  - Restore registers before return
- Caller saved registers (not preserved for caller)
  - Do what you please and expect callees to do likewise
  - Should be saved by the caller if needed after procedure call

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 pipelining we want to overlap instructions in different stages



When pipelining we might run into 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

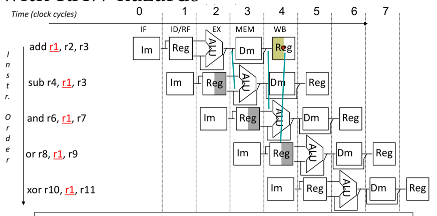
When we have a structural harzard two instructions are trying to use the same hardware within the same cycle, to fix this we can add a delay to all instructions making them the same length and therefore not needing the same resource withing the same cycle.

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. Stalls can have a significan effect on performance. We could 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



Eliminates stalls for dependencies between ALU instrs. 49

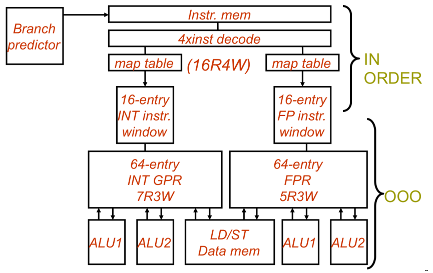
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not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

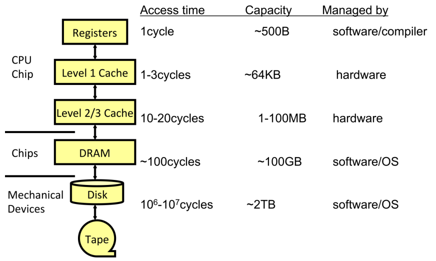
There are some issues with multiple instructions executing at onc,e we need to double the amount of hardware, we introduce hazards, branch delay, & load delay

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



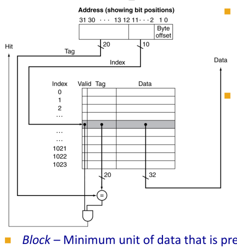
	Access time	Capacity	Managed by
Registers	1cycle	~500B	software/compiler
Level 1 Cache	1-3cycles	~64KB	hardware
Level 2/3 Cache	10-20cycles	1-100MB	hardware
DRAM	~100cycles	~100GB	software/OS
Disk	10^6-10^7cycles	~2TB	software/OS

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



- Assumptions
  - 32-bit address
  - 4 Kbyte cache
  - 1024 blocks, 32 bit/block
- Steps
  1. Use index to read V, tag from cache
  2. Compare read tag with tag from address
  3. If match, return data & hit signal
  4. Otherwise, return miss

- *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

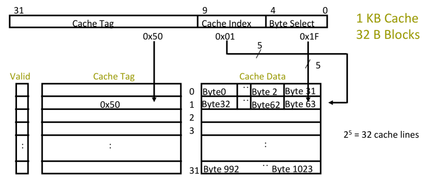
Cheat sheet
by Kenny Luu, page 2 of 2

Cache Blocks

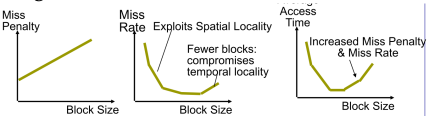
Assuming a 2^nbyte direct mapped cache with 2^m byte blocks

- 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

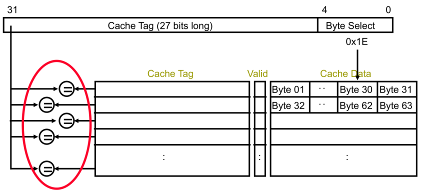


Block Sizes: Larger block sizes take advantage of spatial locality, however it also incurs larger miss penalty since it takes longer to transfer the block into the cache.



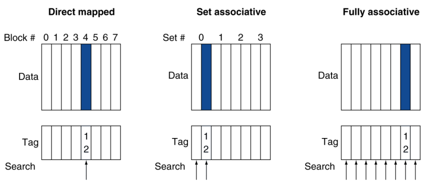
Fully Associative Cache

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

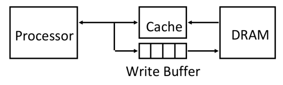
Organization	# of sets	# blocks / set	12 bit Address
Direct mapped	16	1	tag 4, index 4, blk off 4
2-way set associative	8	2	tag 5, index 3, blk off 4
4-way set associative	4	4	tag 6, ind 2, blk off 4
8-way set associative	2	8	tag 7, i 1, blk off 4
16-way (fully) set associative	1	16	tag 8, blk off 4

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:

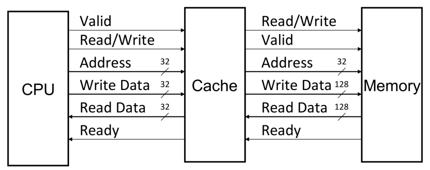
	Write through				Write back	
	Write allocate		No write allocate		Write allocate	
Steps	fetch on miss	no fetch on miss	write around	write invalidate ONLY DM CACHE	fetch on miss	no fetch on miss
1	pick replacement	pick replacement			pick re-placement	pick re-placement
2				invalidate tag	[write back]	[write 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.

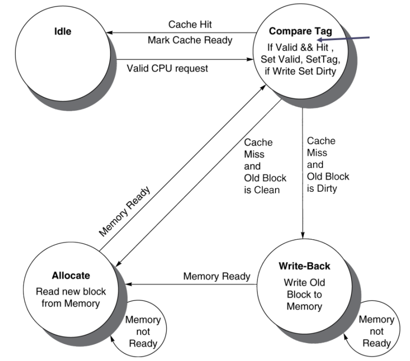
Multilevel Caches

- Primary (L1) caches attached to CPU
  - Small, but fast
  - Focusing on hit time rather than miss rate
- Level-2 cache services misses from primary cache
  - Larger, slower, but still faster than main memory
  - Unified instruction and data (why?)
  - Focusing on low miss rate rather than low hit time (why?)
- Main memory services L2 cache misses
  - Many chips include L3 cache

Interface Signals



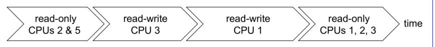
Cache Controller FSM - Write


Idle
Cache Hit
Mark Cache Ready
Compare Tag
If Valid && Hit , Set Valid, SetTag, if Write Set Dirty
Valid CPU request
Cache Miss and Old Block is Clean
Cache Miss and Old Block is Dirty
Memory Ready
Allocate
Read new block from Memory
Memory not Ready
Memory Ready
Write-Back
Write Old Block to Memory
Memory not Ready

Cache Coherence Protocol

Divide lifetime of a memory value into epochs (time periods)

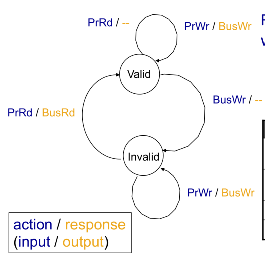
1. Single-Writer, Multiple-Read (SWMR) Invariant
   For any memory location A, at any given time (epoch), there exists only a single CPU that may write to A (and can also read it) or some number of CPUs that may only read A
2. Data-Value Invariant
   The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read–write epoch



Implementation of Cache Coherency Protocols

- Protocols rely on monitoring core actions
  - For now, assume all misses visible to all cores
    - Interconnect makes all actions visible
    - Cache FSM also reacts to requests from other caches
  - Known as a **snooping-based** protocols
- Protocols avoid stale copies
  - Whenever a core becomes a writer: invalidate copies

Simple Coherence Protocol



Action	Abbreviation
Processor Read	PrRd
Processor Write	PrWr
Bus Read	BusRd
Bus Write	BusWr

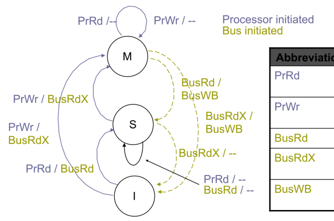
2-State Protocol Coherency

- Assume bus transactions and memory operations are atomic
  - Processor waits for memory operation to finish before issuing next
  - With one-level cache, assume invalidations applied during bus xaction
- SWMR Invariant
  - All writes go to bus + atomicity, causes transition to invalid state
- Data-Value Invariant
  - Read misses appear on bus, and will "see" last write in bus order
  - Read hits: do not appear on bus
  - But value read was placed in cache by either
    - Most recent write by this processor or most recent read miss
    - Both these transactions appeared on the bus
  - So reads hits also see values as produced bus order

3-State Write-Back Invalidation Protocol

- 2-State protocol
  - + Simple hardware and protocol (used in Sun Niagara)
  - - Bandwidth: write-through, every write goes on bus
    - Can work in multi-core with additional level of shared cache
- 3-State protocol (MSI)
  - Modified (called Exclusive in some books)
    - One cache has valid/latest copy
    - Memory is stale
  - Shared
    - One or more caches (and memory) have valid copy
  - Invalid
- Must invalidate all other copies before entering modified state
  - Requires bus transaction (order and invalidate)

MSI Coherence Protocol



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