CP For CSC3050

Chapt 1 Intro

$P_{dynamic} = 0.5 \times C_L \times V_{dd}^2 \times f_{switching}$

ISA- Instruction Set Architecture

ABI- combine of ISA and os interface

Chapt 2 Digital Logic

Radix = Base = max digits in number system

Non-Decimal Conversion: x -> binary -> y

Decimal Conversion: 0x4AF = (16^2)*4 + (16^1)*10 + (16^0)*15

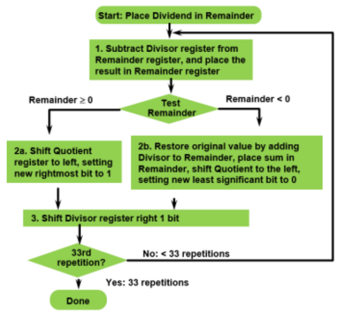
2's complement: 00000010 -> 11111110

Fan in – the number of inputs to the gate

Fan out – the number of gates driven by output, has upper bound.

Combinational logic – Instance

Sequential logic – Latch, flip-flop, registers, etc.


Start: Place Dividend in Remainder
1. Subtract Divisor register from Remainder register, and place the result in Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift Quotient register to left, setting new rightmost bit to 1
2b. Restore original value by adding Divisor to Remainder, place sum in Remainder, shift Quotient to the left, setting new least significant bit to 0
3. Shift Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

Chapt 3 ISA

CISC – Complex Instruction Set Computer

RISC – Reduced Instruction Set Computer

MIPS Registers holds 32x32-bit regs, 2 read port + 1 write port

It is faster than main memory, easier for complier to use, can hold variables.

Add a, b, c -> a = b + c

32-bit = 4 bytes = 1 word

$tx for temporaries, $s for saved, $gp for global pointer, $sp for stack pointer, $fp for frame pointer, $ra for return address, $ax for arguments, $vx for returned values

MIPS is Big Endian, starting from MSB, left to right

Registers are faster than memory.

NO Subtract imm instruction. Only add unary.

Often use "add $s2 $s1 $zero" to copy $s2 from $s1

All use sign-extend, except for andi, ori, xori for 0-extend.

R-format:

Op 6-bit, rs rt rd shamt 5-bit, funct 6-bit, left to right.

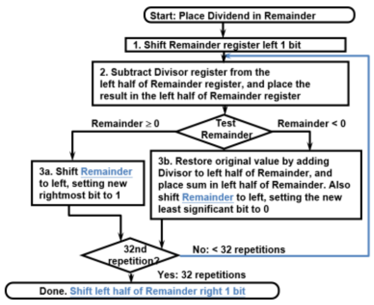
I-format:

Op 6-vit, rs rt 5-bit, const/addr 16-bit, get/lod into rt from addr(rs+imm)

Sll, srl, sla: fill with 0; sra fill with MSB digits.

J: absolute position(imm * 4 + 4)

Beq/bne: relative position(PC+imm*4 + 4)


Start: Place Dividend in Remainder
1. Shift Remainder register left 1 bit
2. Subtract Divisor register from the left half of Remainder register, and place the result in the left half of Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
3a. Shift Remainder to left, setting new rightmost bit to 1
3b. Restore original value by adding Divisor to left half of Remainder, and place sum in left half of Remainder. Also shift Remainder to left, setting the new least significant bit to 0
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done. Shift left half of Remainder right 1 bit

Basic block: a sequence of instructions with no branch/embedded branch

Slt: rd rs rt, if(rs<rt) rd=1, else 0.

Slti: rt rs imm, if(rs<imm) rt=1, else 0.

In memory: 0~0x00400000:Reserved, 0x00400000~0x10000000:Text, 0x10000000~0x7fffffff: static with $gp at button growing upwards, dynamic data starts at where static data ends, stack at the top growing downwards with $sp.

$pc points at the start of text part, i.e. 0x00400000.

Non-Leaf Procedures: procedures that call other procedures, for nested call, caller need to save on the stack. Restore from the stack after the call.

Lui rt, constant: copy 16-bit constant to left(upper) 16 bits of rt, clears right 16 bits of rt to 0.

Chapt 4 ALU

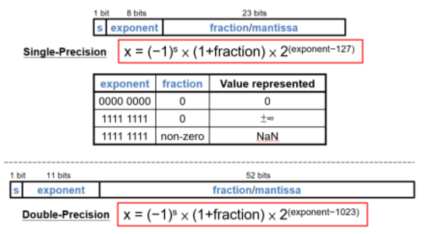
Overflow if (carry in MSB[1] == carry in MSB[2]) and (carry out MSB = carry in MSB[1]).

Or we can do: Overflow = CarryIn[N-1] XOR(^) CarryOut[N-1]

Zero Detection logic uses a big NOR logic gate.

Ripple Carry Adder may have delay of 2n+1, but Carry-Lookahead Adder has only 4 delay.

For multiplication, double precision product is produced.



$x = (-1)^s \times (1+\text{fraction}) \times 2^{(\text{exponent}-127)}$

exponent	fraction	Value represented
0000 0000	0	0
1111 1111	0	±∞
1111 1111	non-zero	NaN

$x = (-1)^s \times (1+\text{fraction}) \times 2^{(\text{exponent}-1023)}$

And it may produce more than 32 bits of output: The result will be stored in two extra registers: HI for MSB-32-bits, LO for LSB-32-bits.

And their value is fetched by two instructions: mfhi rd/mflo rd.

Mult/Multu: put the all 64 bits to LO/HI. Mul rd, rs, rt: put LSB-32 bits of product rs*rt to rd, discard MSB-32-bits.

MIPS Div: div $t1 $t2 # t1/t2.Quotient stored in Lo, remainder in Hi. It will not provide overflow / divide-by-0 check.

Signed Divide: -7/2 = -3, rem = -1; -7/-2 = 3, rem = -1. (Satisfy Dividend = Quotient*Divisor + Remainder)

Floating-point numbers: Defined by IEEE 754-1985 standard. Two representations: Single precision(32-bit), Double precisions(64-bit). Fraction part is in binary form. When being multiplied by 2 x times, the digit move right for x digits. Now consider a 4-digit binary example: $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375), (1)Align binary points: $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$. (2) Add significands: $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$. (3) Normalize result & check for over/underflow: $1.000_2 \times 2^{-4}$, with no over/underflow, (4) Round and renormalize if necessary

In MIPS, Floating Point adder is much more complex than int adder, would possibly take several cycles.(Can be pipelined) Also, FP adder is the same complex as FP mul. We separate registers $f0/$f1,… to load Floats. FP instructions operate only on FP registers, don't do int ops on FP data, vice versa. Instructions: add.s(single), add.d(double),…

Chapt 5 Datapath

CPU Performance factor: Instruction count, CPI(clock cycle per instruction).

$t_{prop} + t_{combinational} + t_{setup} + t_{skew}$ = the minimum time between two clocks. T_setup: the time before the rising clock edge that the input to a flip-flop must be valid. T_skew: difference in absolute time between when two state elements see a clock edge.

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

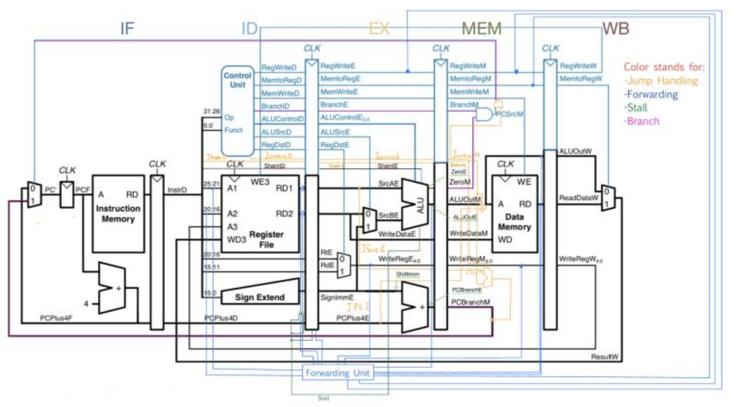
Chapt 6 Pipeline

Five stages: IF-Instruction fetch from memory, ID-Instruction decode & register read, EX-Execute operation or calculate address, MEM-Access(data) memory operand, WB-Write the result back to register.

For single cycle datapath, we only uses "necessary" steps, not expand all to 5 stages.P

How to solve R-type's write? Delay it, make every instruction 5 stages by inserting NOP stage. Sw: WB-NOP, BEQ: MEM+WB-NOP.

Type of hazards: **Structural hazards**: Attempt to use the same resource in two different ways at the same time. **Data hazards:** Attempt to use item before ready. **Control hazards:** Attempt to make decisions before condition is evaluated.



We can always handle hazard by waiting.

Structure hazards: Solution: separate instruction/data memories, make data memory write(MEM) in in the first half, exact/read data from memory(IF) in the second half.

Data hazards: Type of data hazards: (1)RAW, <u>i2 try to read operand before i1 writes it</u>,(2)WAR (3)WAW. However, (2) and (3) cannot happen in MIPS 5-stage pipeline because all instructions are of 5 stages and it's of equal clock time.

How to handle data hazards: (1) Inserting 3(maybe) NOPs: delay i2 for two clock cycles, i.e., insert two bubbles. (2) Forwarding, R-Type-use.

IF	ID	Exec	MEM	WB

- Hazard conditions
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt } Forward from EX/MEM pipeline registers
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt } Forward from MEM/WB pipeline registers

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.RegisterRs1))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
```

Two optimizations

- Do not forward if instruction does not write register
  => check if RegWrite is asserted
- Do not forward if destination register is $0
  => check if RegisterRd = 0

For R-type(First), often rd, rs, rt.

For I-type(Second), often rt, imm(rs)

For beq/bne, often rs, rt, label

For J-type, often j label.

Forwarding - Input to ALU from any pipe register

- Add multiplexors to ALU input
- Control forwarding in EX => carry Rs in ID/EX

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Control signals for forwarding

- EX hazard

```
• if (EX/MEM.RegWrite
   and (EX/MEM.RegRd≠0)
   and (EX/MEM.RegRd=ID/EX.RegRs)) ForwardA=10
```

(3) Stalls, load-use(hardware). Solution: Insert 1 bubble + forwarding. Affected range: next/next-next instruction uses registers ready to be loaded.

and its use

if (ID/EX.MemRead and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.registerRt))) stall the pipeline for one cycle

How to stall? Not change PC and IF/ID, insert an NOP by changing EX, MEM, WB control fields of ID/EX pipeline register to 0.

Handling branch hazards: (1) Assume branch not taken, add hardware for flushing. Branch decisions made at MEM -> need to flush instructions in IF/ID, ID/EX. (2) Reduce delay of taken branch by moving branch execution earlier in pipeline, make branch check at ID, so that we only need to flush one instructions. Add a control signal called IF.Flush. (3) Dynamic branch prediction. (4) Compiler rescheduling, delay branch.

Handling Exceptions: Before jump to OS, save PC to offending(or interrupted) instruction. In MIPS, it's Exception Program Counter(EPC), 32 bit. Also save indication of the problem. In MIPS, store in Cause register, 32 bit. Some indicate undefined opcode, some indicate overflow. Also, for arithmetic overflow, jump to handler at 0x8000 00180. For undefined instruction, jump to handler at 0x8000 00000. In OS, read cause, and transfer to relevant handler, determine action required. If restartable, take corrective action and use EPC to return to the program. Otherwise, terminate program and report error using EPC, cause, …. <u>For exceptions in Pipeline</u>, if "add $1, $2, $1" occurs overflow, then prevent $1 from being clobbered, complete previous instructions and flush add and subsequent instructions, set cause and EPC register values, transfer controls to handler. This is similar to mispredicted branch, use much of the same hardware. <u>For multiple expectations,</u> the simple approach is to deal with the exception from the earliest instruction, do flush. This can get "precise" exceptions. However, in complex pipelines, it's difficult to be "precise".

For Imprecise Exceptions, just stop pipeline and save state, including exception causes, then let the handler work out, decide whether to flush or continue.

Superscalar and dynamic pipelining, we executing multiple instructions in parallel. To increase <u>instruction parallelism</u>(ILP), we need deeper pipeline(less work per stage => shorter clock cycle). We start multiple pipeline stages per clock cycle, then CPI(<u>clock cycle per instruction</u>) < 1, so we use IPC(<u>Instructions Per Cycle</u>). 1/CPI = IPC. <u>Errors</u> can be (1) *static multiple issue*(when compiling), compiler groups instructions to be issued together, packages them into "issue slots", compiler detects and avoid hazards. For static multiple issues, complier group instruction into issue packets, which is the group of instructions that can be issued on a single cycle. It's determined by pipeline resources required. Above common ones are called one-issue packets. If two, then two instructions get into two Ifs at the same time. Also, pad an unused instruction with nop. For EX hazards, now we cannot place add and lw(with conflict) in the same packets. We split into two packets, effectively a stall. For Load-use hazard, still we delay by one cycle and with bubble. Some problems: replicate loop body to expose more parallelism. We should reduces loop-control overhead. Also, when looping, we should use different registers per replication, which is also called "register renaming", to avoid loop-carried "anti-dependencies".

(2)*Dynamic multiple issue*(when executing), CPU examines instructions stream and chooses instructions to issue each cycle, compiler can help by reordering instructions, CPU resolves hazards using advanced techniques at runtime. Pipeline is divided into 3 major units: (1) an instruction fetch and decode unit, (2) many functional units, (3) a commit unit. Functional unit and commit unit have buffer, to store the operand and store the result. For predictions, do not commit until branch/load prediction cleared/determined. Why dynamic scheduling? Because not all stalls are predicable(cache miss), cannot always schedule around branches, different implementations of an ISA have different latencies and hazards.

Chapt 7 Memory and cache

Random access: SRAM(Static Random Access Memory), low density, high power, fast, expensive, static(until lose power), for caches. 6-8 transistor and 2 resistors as one cell.

DRAM(Dynamic Random Access Memory), high density, low power, cheap, slow, Dynamic(has to reflash time to time), for main memory. 1 transistor and 1 capacitor for a cell.

SDRAM(Synchronous DRAM), read/write needs synchronization.

DDR SDRAM(Dual Date Rate SDRAM), data transfer on both rising/falling clock edge.

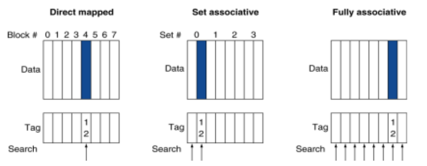
Flash Memory: Will not lose data when cut power. Electrically erasable programmable read-only memory(EEPROM)

Magnetic Disk and Tape

Block: basic unit of info transfer. Unit of copying. Minimum unit of information that can either be presented or not presented in a level of the hierarchy. Temporal locality (时间位置)，Spatial locality（空间位置）

If accessed data is present in upper level, Hit. Otherwise Miss. Hit time << Miss penalty. Registers <-> Memory: by compiler(programmer?), Cache<-> Memory, by hardware. Memory <-> Disks, by hardware and OS, also programmer.

Three kinds of associative cache level:



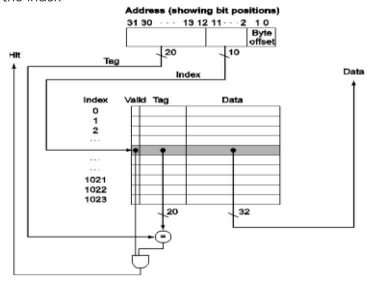
Address mapping: (block address) mod (number of blocks in cache). Valid bit could 0/1.

Byte offset: 在这里指的是在这个 4 bytes block（1 word）中处于第几个 block(byte)。index 由 cache 的大小决定，tag 即余下部分。

Calculate formula:

If cache size is 2^n block, then n bits are used for the index

If the block size is 2^m words (2^(m+2) bytes), then m bits are used for the word within the block. The size of the tag field is : 32 – (n+m+2).The total number of bits in a directed mapped cache: 2^n * (block size + tag size + valid field size).



Larger blocks should reduce miss rate should reduce miss rate, but with fewer number of blocks, more competition => increase miss rate. Therefore it's a tradeoff.

Cache read: (1) read hit- process normally, (2) read miss- stall CPU pipeline, fetch from next level. If instruction cache miss, restart IF. If data cache miss, wait until complete data access.

, Cache write: (1) write hit (a) Write-through- Also update data in memory. Pros: ensure fast retrieval while making sure data is not lost and in backing store. Cons: experience latency(delay). Solution: Write buffer, hold data waiting to be written to memory. Detail: Processor -> Cache and WB, while WB -> DRAM by memory controller. WB is FIFO. Typical entries:4. If Store frequency > 1/DRAM write cycle, It will stall since WB is already full.(读比写快）, (b) write-around, data is written only to the backing store without writing to the cache. Pros: Good for not flooding the cache with data that may not subsequently be re-read. Cons: Reading recently written data will result in a cache miss (and so a higher latency) because the data can only be read from the slower backing store.

(c) write-back, Just update the block in cache, keep track of "dirty" blocks. When a dirty block is replaced, write it back to the memory. This cause inconsistency in cache and memory. Pros: low delay(latency) and high throughput for write-intensive applications. Cons: Risk of cut power- cache failure, data loss. (2) write miss- for write-through, (a) Write allocate: block fetched from memory, also block in memory is overwritten. (b) No write allocate: only fetch from memory. As for write-back, usually use write allocate.

How many total bits are required for a direct-mapped cache with 16 KiB of data and 4-word blocks, assuming a 32-bit address?

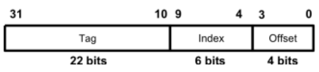
16KB is 4096 (2^12) words, with a block size of 4 words, there are 1024 (2^10) blocks:
For each block
- Data part: 4*32 bits
- Tag part: 32-10-2-2 = 18 bits
- Valid bit: 1 bit
- Total: 2^10 * (4*32 + 18 + 1) = 147 Kb

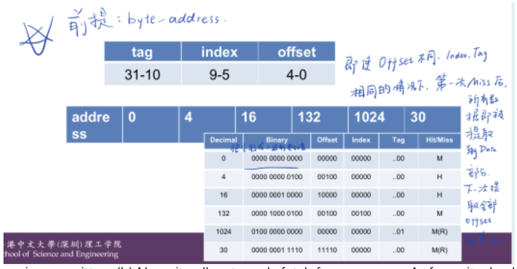
For a cache with 64 blocks, 16 bytes/block
- To what block number does address 1200 map?
- What is the length of the tag field?

Block address = ⌊1200/16⌋ = 75

Block number = 75 modulo 64 = 11

Fields are as follows




前提：byte-address.
即使 Offset 不同，Index, Tag 相同的情况下，第一次 Miss 后，所有数据即被提取到 Data 部分，下一次提取全部 Offset 都是 hit
港中文大學(深圳)理工学院
School of Science and Engineering

tag	index	offset
31-10	9-5	4-0

address	0	4	16	132	1024	30

Decimal	Binary	Offset	Index	Tag	Hit/Miss
0	0000 0000 0000	00000	00000	..00	M
4	0000 0000 0100	00100	00000	..00	H
16	0000 0001 0000	10000	00000	..00	H
132	0000 1000 0100	00100	00100	..00	M
1024	0100 0000 0000	00000	00000	..01	M(R)
30	0000 0001 1110	11110	00000	..00	M(R)

Memory support:

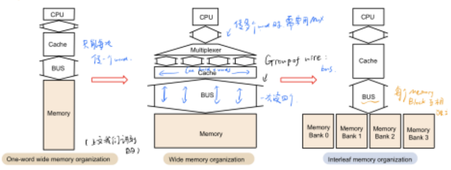
Miss penalty:

One-word-wide: 1+4*15+4*1

Four-word-wide: 1+15+1

Four-bank, one-word-wide: 1+1*15+4*1

How to increase memory bandwidth to reduce miss penalty?



CPU time = (Program execute cycles + memory stall cycles)*clock cycle time.

Memory stall time = memory access time * miss rate * miss penalty.

I-cache: about registers, every instructions will call it. D-cache: about memory. Only be called when lw/sw called. Often calculate by CPI to compare performance.

AMAT(Average memory access time) = Hit time + miss rate*miss penalty = %instr x (instr hit time + instr miss rate x instr miss penalty) + %data x (data hit time + data miss rate x data miss penalty).

Another Understanding: AMAT = time spent in hit + time spent in miss = cache hit time * cache hit rate + miss time * miss rate.

Associate Caches: diminishing returns(边际效用递减)

- Q: Assuming a cache of 4096 blocks, a 4-word block size, and a 32-bit address, find the total number of sets and the total number of tag bits for caches that are direct mapped, two-way and four-way set associative, and fully associative.
- For 2-way associative, total tag bits:
  - # sets = 2048
  - Byte offset:2, word offset: 2, index bits: 11, so the tag bits are 32-11-2-2 = 17
  - Total tag bits are 17 *2 *2048 = 68K bits.
- For 4-way associative, total tag bits:
  - # sets = 1024
  - Byte offset:2, word offset: 2, index bits: 10, so the tag bits are 32-10-2-2 = 18
  - Total tag bits are 18 *4 *1024 = 72K bits.
- For full associative, only 1 set, total tag bits are 28 *4096 = 112K bits.

Cache index = (block address) mod (number of sets)

Data placement policy: if misses, we need to decide which block to throw out. (1)Random, (2) LRU(Least Recently Used), uses a pointer to track or track by increasing index.

N-Way set-associative cache: Data comes after hit/miss decision and set selection. Direct mapped cache: cache block is available before Hit/miss.

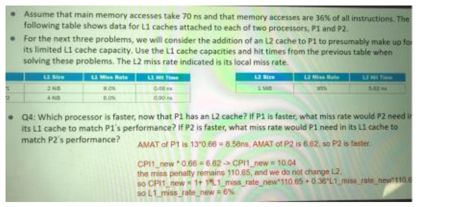
Multilevel Cache AMAT:

AMAT = L1 hit time + L1 miss rate * L1 miss penalty. (L1 miss penalty = L2 hit time + L2 miss rate* L2 miss penalty)

- Given(1GHz = 1ns, 4GHz = 0.25ns)
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 × 400 = 9

Chapt 8 Virtual Memory

Diagram: New->admitted->ready; waiting -> I/O or event completion -> ready; ready -> scheduler dispatch -> running; running -> interrupt -> ready; running -> exit -> terminated; running -> I/O or event wait -> waiting.


- Assume that main memory accesses take 70 ns and that memory accesses are 36% of all instructions. The following table shows data for L1 caches attached to each of two processors, P1 and P2.
- For the next three problems, we will consider the addition of an L2 cache to P1 to presumably make up for its limited L1 cache capacity. Use the L1 cache capacities and hit times from the previous table when solving these problems. The L2 miss rate indicated is its local miss rate.

	L1 Size	L1 Miss Rate	L1 Hit Time
P1	2 KiB	8.0%	0.66 ns
P2	4 KiB	6.0%	0.90 ns

L2 Size	L2 Miss Rate	L2 Hit Time
1 MiB	95%	5.62 ns

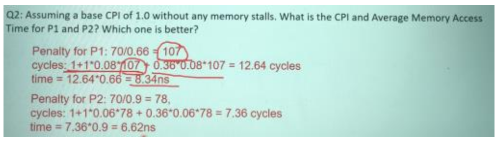
- Q4: Which processor is faster, now that P1 has an L2 cache? If P1 is faster, what miss rate would P2 need in its L1 cache to match P1's performance? If P2 is faster, what miss rate would P1 need in its L1 cache to match P2's performance?

AMAT of P1 is 13*0.66 = 8.58ns, AMAT of P2 is 6.62, so P2 is faster.

CPI1_new * 0.66 = 6.62 -> CPI1_new = 10.04
the miss penalty remains 110.65, and we do not change L2,
so CPI1_new = 1+ 1*L1_miss_rate_new*110.65 + 0.36*L1_miss_rate_new*110.6
so L1_miss_rate_new = 6%

VM can help protect from other programs.

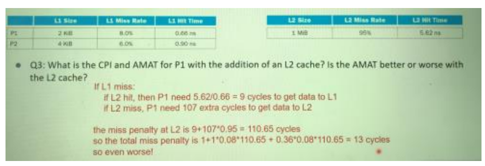
VM "blocks" is called "page", VM translation "miss" is called "page fault". It has huge miss penalty. So we should try best to avoid it. We use fully associative place + use "page table" to locate pages.


Q2: Assuming a base CPI of 1.0 without any memory stalls. What is the CPI and Average Memory Access Time for P1 and P2? Which one is better?

Penalty for P1: 70/0.66 = 107
cycles: 1+1*0.08*107 + 0.36*0.08*107 = 12.64 cycles
time = 12.64*0.66 = 8.34ns

Penalty for P2: 70/0.9 = 78,
cycles: 1+1*0.06*78 + 0.36*0.06*78 = 7.36 cycles
time = 7.36*0.9 = 6.62ns

Page table stored in main memory. Array of page tables, indexed by virtual page number, to find physical page. If page is present in memory, Page table entry(PTE) stores the physical page number. (plus other status bits). If NOT present, PTE can refer to location in swap space on disk. Swap space: the reserved disk space for all virtual space of a process. Hardware must trap to the operating system so that it can remedy the situation. (1)Pick a page to discard (may write it to disk),(2)Load the page in from disk,(3)Update the page table(4),Resume to program so HW will retry and succeed! It will only fetch from Disk when page fault occurs.



	L1 Size	L1 Miss Rate	L1 Hit Time
P1	2 KiB	8.0%	0.66 ns
P2	4 KiB	6.0%	0.90 ns

L2 Size	L2 Miss Rate	L2 Hit Time
1 MiB	95%	5.62 ns

- Q3: What is the CPI and AMAT for P1 with the addition of an L2 cache? Is the AMAT better or worse with the L2 cache?

If L1 miss:
if L2 hit, then P1 need 5.62/0.66 = 9 cycles to get data to L1
if L2 miss, P1 need 107 extra cycles to get data to L2

the miss penalty at L2 is 9+107*0.95 = 110.65 cycles
so the total miss penalty is 1+1*0.08*110.65 + 0.36*0.08*110.65 = 13 cycles
so even worse!

To reduce page fault rate, prefer LRU(least recent used). We also uses write-back, write it in the page, and set dirty bit in PTE.

To solve problem that PT is too big, we use hashing, bounds registers, let pages grow in both directions,…

TLB(Translation Lookaside Buffer), helps us to faster locate Physical address, faster than PTE. If TLB miss, then it will go into PTE for further search. If miss again, then go to the disk to find Physical address from virtual address. If address appears in TLB, it must first stored in PTE. Vice not versa.(Subset).

Direct map(1 way) -> N-way set associative -> Fully associative. For Virtual memory, only write-back is feasible, given disk write latency. 3C model: Compulsory misses(cold start), Capacity misses(cache full), Conflict misses(in non-fully associative cache).

Design change	Effects on miss rate	Possible effects
size ↑	capacity miss ↓	access time ↑
associativity ↑	conflict miss ↓	access time ↑
block size ↑	spatial locality ↑	miss penalty ↑

We can use a finite state machine to sequence control steps. If there are more than 1 CPU, there may be cache coherence(多核同时工作，一核 cache 改变而另一核不知情，这时候要求发出一个 signal，告知其他核主动清空各自的 cache,下次从 memory 直接调取).

Chapt 9 I/O system

I/O causes asynchronous interrupts（异步）, controlled by I/O controller hardware.(Transfer data to/from device, synchronize operations with software). Common registers: cause device to do something, Status registers: Indicate what the device is doing and occurrence of errors., Data registers.

BUS: shared communication channel, we need interconnections between CPU, memory, I/O controllers.

- Processor-Memory buses
  - Short, high speed
  - Design goal is to match memory organization
- I/O buses
  - Longer, allowing multiple connections
  - Specified by standards for interoperability
  - Connect to processor-memory bus through a bridge

Reliability: mean time to failure(normal working time, MTTF), Service interruption(error time, MTTR), Mean time between failures(MTBF), MTBF = MTTF + MTTR. Availability: MTTF / (MTTF+MTTR). Increase MTTF or reduce MTTR.

Amdahl's Law: calculate overall speedup caused by certain component.

$$S = \frac{1}{(1-f) + \frac{f}{k}}$$

S is the overall speedup;
f is the fraction of work performed by a faster component;
k is the speedup of the faster component.

I/O can be controlled in four general ways

(1)Programmed I/O (PIO), it requires polling(轮询), which is of low efficiency. Acceptable in some small gargets. including,(a)Reserve a register for each I/O device and each register is continually polled to detect data arrival ,(b)CPU wait -> detect "data ready" -> act as programmed instructions

(2)Interrupt-driven I/O , only runs when interrupt occurs, faster than above method. Including:,(a)Allow CPU to do other things until I/O is requested, (3)Direct memory access (DMA): Offloads I/O processing to a special-purpose chip that takes care of the details. Method 1 and 2 need to transfer data via CPU, which is of low efficiency. Now DMA can directly access memory, which becomes faster. OS provides starting address in memory, I/O controller transfer to/from memory autonomously. However, there are some problems: In a system with caches, there can be two copies of a data item, one in the cache and one in the main memory. For a DMA input (from disk to memory) – the processor will use stale data if that location is also in the cache. For a DMA output (from memory to disk) and a write-back cache, the I/O device will receive stale data if the data is in the cache and has not yet been written back to the memory. To solve these problems: we can (a)Route all I/O activity through the cache – expensive and a large negative performance impact,(b)Have the OS invalidate all the entries in the cache for an I/O input or force write-backs for an I/O output (called a cache flush),(c)Provide hardware to selectively invalidate cache entries – i.e., need a snooping cache controller.

(4)Channel I/O :Use dedicated I/O processors. It is an intelligent type of DMA interface. Channel I/O consists of one or more I/O processors (IOPs) that control various channel paths. Slower devices such as terminals and printers are combined (multiplexed) into a single faster channel. On IBM mainframes, multiplexed channels are called multiplexor channels, the faster ones are called selector channels.

Magnetic Disk: platter(盘) made of magnetizable material, each unit called "Sector", sectors that formed as a concentric cycle are called "track". Cylinder / Surface / Sector, Seektime 指的是 Arm 水平移动所需时间, rotational delay 指的是磁盘旋转至相应位置所需要的时间。Seek time + rotational delay = access time.

光盘 Optical Disks: 由一个个凹槽 called pit and land 组成，进化历程:CD-ROM(~700MB) -> DVD (~17GB) -> Blu-ray (~25GB)

RAID: Redundant Array of Independent Disks. Cheap, reliable, cost-effective. It can provide error collection.

RAID0: also known as "drive spanning", no redundancy. Low reliability.

RAID1: provides 100% redundancy. High reliability, but low speed.

RAID2: Hamming code used, but neither high reliability nor high speed.

RAID3: dedicated parity(奇偶效验),冗余有限只允许一位错误，适合自用

RAID4: adding parity disks to RAID 0

RAID5: 将 RAID4 的 parity disk 位置放在了对角线上，both high performance and reliability, commercial use.

RAID6: Add new 校验法 called Reed-Soloman, highly fault-tolerant.

Future storage: DNA/Holographic/…