1. Cost vs. Price:
   - Manufacturing Cost: component cos, direct cost, gross margin: R&D, marketing, sales,taxes
   - List Price: include Average Discount of 15% to 35% of Volume discounts and/or retailer makeup
   - Yield: % số components qua được bước testing
   - Volume:
   - Feature Size: minimum size of transistor or wire in x or y direction
2. Manufacturing Stages:
   - Wafer growth & testing
   - Cắt wafer thành dies
   - Đóng gói die thành chips
   - Test chip
3. Cost:
   - $Cost\ of\ IC = \frac{die\ cost + die\ test + packaging + final\ test}{final\ test\ yield}$
   - $Die\ cost = \frac{Cost\ of\ Wafer}{dies\ per\ wafer * dies\ yield}$
   - $Dies\ per\ Wafer = \frac{\Pi*(d_W/2)^2}{diesdrea} - \frac{\Pi*(d_W)}{\sqrt{2*die\ area}}$
   - $Dies\ yield$: tỷ số hoặc % good dies trên một wafer
   - $Wafer\ Yield(WY)$: account for completely bad wafers no need not be tested.
   - $DieYield = WY * (1 + \frac{(Defect/UnitArea)*DieArea}{\alpha})^{-\alpha}$
   - For CMOS: $\alpha = 4$
4. Time:
   - $CPU\ Time$: only computing time, not include waiting time for I/O or running other program.
   - $User\ CPU\ Time$: CPU time spent in the program
   - $System\ CPU\ Time$: CPU time spent in OS performing task requested by the program decrease exec time.
   - $CPU\ Time = User\ CPU\ Time + System\ CPU\ Time$
   - $Execution\ Time$: Time to do task
   - $Throughput$: task per day, hour, week,...
5. Performance:
   - $Performance_A = 1/ExecTime_A$
   - $Speedup = n = \frac{Performance_A}{Performance_B} = \frac{ExecTime_B}{ExecTime_A}$, máy A nhanh hơn máy B n lần.
   - $CPUTime = I * CPI * C$
     - I: số lệnh cần thực thi của program ($inst/program$)
     - CPI: số chu kỳ thực thi 1 lệnh $cycle/inst$
     - C: CPU clock cycle ($= 1/clock\ rate$)
   - $Speedup = \frac{ExecTime_{old}}{ExecTime_{new}} = \frac{I_{old}*CPI_{old}*C_{old}}{I_{new}*CPI_{new}*C_{new}}$
   - Nếu program có n loại lệnh và C_i là số lệnh $type_i$, CPI_i là số chu kỳ trên lệnh $type_i$ thì:
   $CPI = CPU\ clock\ cycles/Inst.Count$
     - $CPU\ clock\ cycles = \sum_{i=1}^{n} CPI_i * C_i$
     - $Inst.Count = \sum_{i=1}^{n} C_i$
   - Nếu program có n loại lệnh và C_i là số lệnh $type_i$, CPI_i là số chu kỳ trên lệnh $type_i$, F_i là % lệnh $type_i$ thì:
   $CPI = \sum_{i=1}^{n} CPI_i * F_i$
     - $F_i = C_i/Inst.Count$
     - $ExecTime\ of\ Type_i = \frac{CPI_i*F_i}{CPI}$
6. Amdahl's Law:
   - $Speedup_{overall} = \frac{1}{(1-F)+F/S}$
     - F: Thành phần cải tiến được
     - S: Tốc độ cải tiến của thành phần tương ứng
   - Ví dụ: RISC CPU có lệnh LOAD chiếm 45% và cần 5 chu kỳ để thực hiện, người ta cải tiến lệnh này chỉ cần 2 chu kỳ. Tính speedup toàn hệ thống.
   Ta có: $F = 45\%$ là thành phần được cải tiến. $S = 5/2 = 2.5$, theo Amdahl's Law thì:
   $Speedup = \frac{1}{(1-S)+F/S} = \frac{1}{(1-.45)+.45/2.5} = 1.37$
   - Tổng quát:
   $Speedup_{overall} = \frac{1}{(1-\sum_{i=1}^{n} F_i)+\sum_{i=1}^{n} \frac{F_i}{S_i}}$
7. MIPS Rating:
   - $MIPSRating = \frac{ClockRate}{CPI*10^6}$
   - $MIPSRating\ DO\ NOT$ account for count of instruction executed ⇒ Higher $MIPSRating\ may\ not$ mean higher performance or better exec time.
   - $MIPSRating\ DO\ NOT$ account for the ISA used ⇒ Cannot be used to compare CPUs with different instruction sets.
   - $MIPSRating$ is valid to compare performance of different CPU if only if:
     - The same program is used
     - The same ISA is used
     - The same compiler is used
8. Important formula:

$$CPI = \frac{CPU_{cc}}{Inst.Count} = \frac{\sum_{i=1}^{n} CPI_i * C_i}{Inst.Count}$$

$$CPU_{Time} = \frac{Inst.Count * CPI}{ClockRate}$$

9. MIPS MCU:
   - Registers: R0-R31, PC, HI, LO
   - Instruction Categories: Load/Store, Computational, Jump & Branch, Floating Point, Memory Management... - Instruction Format: 32-bits wide
     - R-Format: |OP|rs|rt|rd|shamt|funct|
     - I-Format: |OP|rs|rt|immediate|
     - J-Format: |OP|jump target|
   - Register details:

Name	Number	Desc	Saved?
$zero	0	constant 0	n.a
$v0-$v1	2-3	return values	no
$a0-$a3	4-7	arguments	yes
$t0-$t7	8-15	temporaries	no
$s0-$s7	16-23	saved values	yes
$t8-$t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr(HW)	yes

   - Big Endian: trọng số cao nằm ở byte addr thấp
   - Little Endian: trọng số thấp nằm ở byte addr thấp
   - MIPS: thuộc loại Big Endian
10. MIPS Instructions: - $R - Format$:
    - OP-Rs-Rt-Rd-Shamt-Funct, với Rs,Rt thanh ghi nguồn 1 và 2, Rd là thanh ghi đích
    - Gồm các lệnh: số học, logic,...
    - OPCODE luôn bằng 000000
    - ADD, SUB, MUL, DIV, AND, OR, XOR, XNOR, SLL, SRL, SRA *(dịch phải đại số)*,
    - Ví dụ:
      SUB rd, rs, rt ⇒ $rd = rs - rt$
      MUL rd, rs, rt ⇒ $rd = rs * rt$: rd chứa 32 bit thấp của kết quả và không ghi kết quả vào HI và LI
      DIV rs, rt ⇒ $(HI, LO) \Leftarrow rs/rt$

    - $I - Format$:OP-Rs-Rt-16 bit immediate value
      - ADDI, ORI,
      - LOAD/STORE: LW $t0, 16($t1) ⇒ load word ở ô nhớ được chỉ bởi t1 + 16 vào t0
      - Near Branch: BEQ $s0,$s1,256 ⇒ $PC = PC + 4 * 256$
    - $J - Format$: OP-26 bit Offset
      - J: nhảy không điều kiện đến nhãn
      - JAL: gọi thủ tục trong vùng 256MB
      - JR $ra: return

-*Others* :

- SLT rd, rs, rt ⇒*If rs is less than rt, the result is 1 (true); otherwise, it is 0 (false)*
- Atomic Read-Modify-Write: *Load Link Word & Store Conditional Word*
  LL $Rt, offset($Rs)
  SC $Rt, offset($Rs)
  If the content at memory address specified by LL is modified before SC to the same address, SC fails and return 0 in $Rt. Or else, SC store $Rt to memory and return 1 in $Rt.
- Load 32 bit immediate value
  LUI $t0, 0x1234
  ORI $t0, $t0, 0x5678

11. Pipeline

- Pipelining doesn't reduce number of stages:
  - doesn't help latency of single task
  - helps throughput of entire workload
- Speedup phụ thuộc vào số tầng pipeline
- Pipeline speed bị giới hạn tại tầng chậm nhất ⇒ thời gian các tầng không đều nhau sẽ làm giảm speedup
- *Fill time* và *Drain time* làm giảm speedup.
- Nếu tải phụ thuộc vào tải khác thì phải đợi ⇒ *stall, nop, delay*

- MIPS pipeline có 5 tầng:

- IF: Load lệnh từ bộ nhớ lệnh
- ID: Giải mã lệnh, đọc toán hạng trong thanh ghi, tính toán địa chỉ có thể phải rẽ nhánh
- EX: Thực thi lệnh hoặc tính địa chỉ *(tùy vào loại lệnh)*
- MEM: Truy xuất toán hạng từ bộ nhớ dữ liệu
- WB: Ghi kết quả vào thanh ghi

- Chú ý:

- Ghi thanh ghi ở cạnh lên và đọc thanh ghi ở cạnh xuống của xung clock.
- Speedup lý tưởng bằng số tầng pipeline $Speedup = \frac{n*p}{\frac{p}{k*(n-1)}+p} = \frac{n}{\frac{1}{k*(n-1)}+1} \approx k$ - k: số tầng pipeline
  - p: latency
  - n: số lệnh
- Speedup:
  - $Speedup = \frac{CPI_{unpipelined}}{CPI_{pipelined}} * \frac{CC_{unpipelined}}{CC_{pipelined}}$
  - $CPI_{unpipelined} = 1 + Pipelined\,stall\,cycles\,per\,instr.$

12. Forwarding to minimize Data Hazard:
- Kết quả ALU tại tầng EX/MEM và MEM/WB được đưa ngược về ALU inputs
- Nếu phát hiện toán hạng cần dùng tại tác vụ ALU hiện tại là kết quả của tác vụ ALU trước đó thì nó sẽ lấy giá trị forward về chứ không lấy giá trị trong thanh ghi.
- Không phải mọi trường hợp Data Hazard đều được giải quyết bằng forwarding.
- Nếu không có giải quyết structure hazard, không được IF tại MEM nếu MEM có truy xuất bộ nhớ.
- Nếu không có forwarding, ID phải được thực hiện tại WB nếu có phụ thuộc thanh ghi.
- Nếu có forwarding, chỉ stall một chu kỳ sau lệnh LOAD và sử dụng kết quả lệnh load cho lệnh ngay phía sau.

13. Control Hazard:
- Khi thực thi một lệnh rẽ nhánh, PC có thể bị thay đổi hoặc không sang một giá trị khác PC+4. Nếu thay đổi PC gọi là *taken*, nếu không gọi là *not taken* hoặc *untaken*.
- Giảm Branch Penalty:

- *Freeze or Flush*: stall tới khi hướng rẽ nhánh rõ ràng.
- *Predicted Untaken*: giả định không rẽ nhánh, load lệnh kế tiếp tại PC+4
- *Predicted Taken*: giả định rẽ nhánh, load lệnh tại đích đến sau khi ID (stall 1 chu kỳ)
- *Delay branch*:
  - *A*: From before branch: lấy lệnh không phụ thuộc bỏ vào $delay - slot$ (best choise). Nếu không thể, chọn *B* hoặc *C*
  - *B*: From branch target: lấy lệnh tại đích đến bỏ vào $delay - slot$ theo kiểu taken
  - *C*: From fall through:lấy lệnh tại đích đến bỏ vào $delay - slot$ theo kiểu untaken
  - $Speedup_{pipeline} = \frac{Pipepline_{depth}}{1+BranchFreq*BranchPenalty}$

14. Caches:
- REG ↔ CACHE ↔ MEM ↔ DISK ↔ TAPE
- 2 tiêu chí dự đoán khi tham chiếu bộ nhớ: thời gian (loop, reuse), không gian (straightline code, array)
- *Cache hit*: dữ liệu CPU cần có sẵn trong cache, *Cache miss*: cần thay thế môt block từ MEM và trả dữ liệu cho CPU.
- Truyền dữ liệu giữa CACHE và MEM do *DMA* đảm trách.
- *Hit rate*: fraction of cache hit. *Miss rate*: $1 - Hit\,rate$
- *Miss Penalty*: Time to replace block + Time to deliver data to CPU
- Replacement:

- Thay thế block nào trong cache? → *Block placement strategy & Cache Organization: Full Associative, $k-way$ Set Associative, Direct Map*
- Làm sao tìm ra một block trong cache? → *Block Identify: Tag/Block*
- Khi cache miss thì thay thế block nào? → *Block replacement: Random, LRU(Least Recently Used), FIFO*
- Khi write thì làm sao? → *Cache Write Policy: Write through, Write back*

- Direct Mapped Cache: N bits 2^N *words*

- Cache có 2^k lines (blocks)
- Mỗi lines có 2^b words
- Block M được map vào lines thứ $M\%2^k$
- Cần $t = N - k - b$ tag bits để xác định Mem block
- Đơn giản nhưng Miss Rate cao

- Fully Associative Cache:

- CAM: Content Addressable Memory
- Mỗi block được map vào bất kỳ lines nào trong cache
- Cần $t = N = b$ tag bits
- Cần $t = N-k-b$ tag bits để xác định Mem block, compare tag với mọi lines
- Chỉ cần thay thế khi hết lines nhưng tốn tài nguyên, delay do phải so sánh với chiều phần tử.

- $w-way$ Set Associative Catch:

- Cân bằng giữa 2 phương pháp trên
- Cache được chia làm 2^k sets
- Mỗi set có 2^w lines
- Block M được map vào bất kỳ line nào trong set $M\%2^k$
- Số tag bit: $t = N - k - b$

- Cache Miss:

- Compulsory Misses: lần đầu tiên access
- Capacity Misses: không đủ chỗ chứa toàn bộ blocks của chương trình thực thi
- Conflict Misses: block bị thế bởi block khác, sau đó nó lại được dùng → *ping pong effect*

- Khi write thì sao?

- Hit: WT ghi cả vào cache và mem. WB chỉ ghi vào cache và ghi mem khi bị replace
- Miss: *no write allocate*: chỉ ghi vào mem, *write allocate* load vào cache rồi ghi.