Kiến trúc máy tính

1. **Chi phí mạch**:

$$CostPerDie = \frac{CostPerWafer}{DiesPerWafer \times Yeild}$$

$$DiesPerWafer \approx \frac{WaferArea}{DieArea}$$

$$Yeild = \frac{1}{1 + (DefectsPerArea \times DieArea/2)}$$

2. **Instruction Set Design**: Thiết kế tập lệnh là quan trọng của hệ thống máy tính.
   a. Những vấn đề quan trọng nhất là:
      _ Bao nhiêu & loại tác vụ sẽ có; độ phức tạp của tác vụ
      _ Kiểu dữ liệu
      _ Định dạng của lệnh: độ dài, số lượng các địa chỉ, kích thước của các trường khác nhau, …
      _ Số lượng các thanh ghi
      _ Mô hình đánh địa chỉ
   b. Các vấn đề này có quan hệ với nhau
3. **Đánh giá hiệu năng CPU:**
   a. 2 thước đo là:
      _ **Response time** (execution time) là thời gian từ lúc bắt đầu → khi 1 task hoàn tất. 1 task gồm truy xuất đĩa, hoạt động I/O, OS overhead, CPU execution time… Quan trọng với ***user***
      _ **Throughput** (**bandwidth** thông lượng) tổng số lượng công việc hoàn tất trong 1 thời gian cụ thể. Quan trọng với hệ thống quản lý dự liệu trung tâm như server.
   b. ***Response time & Throughput bị ảnh hưởng bởi***:
      _ *Thay đổi CPU nhanh hơn*: tăng Response time & Throughput
      _ *Tăng số lượng CPU*: tăng Throughput
4. **Hiệu năng Performance (Speed)**:
   a. Hiệu năng lớn khi thời gian thực thi nhỏ nhất:

$$performance_X = \frac{1}{excecution\_time_X}$$

   b. Nếu X chạy nhanh gấp n lần Y thì

$$\frac{performance_X}{performance_Y} = \frac{excecution\_time_Y}{excecution\_time_X} = n$$

   c. Giảm thời gian đáp ứng (response time) hầu như luôn luôn cải thiện thông lượng (throughput).
5. **Đo hiệu năng**:
   a. ***Elapsed time*** (***wall clock time***, ***respone time***) là tổng thời gian đáp ứng & các yếu tố khác (processing, I/O, OS overhead, idle time). Dùng để xác định hiệu năng của hệ thống
   b. ***CPU time*** (ko kể thời gian truy xuất ổ cứng, I/O): là thời gian CPU thực thi 1 task. Bao gồm thời gian CPU cho người dùng & cho hệ thống. Dùng xác định hiệu năng của CPU
   c. *Different programs are effected differently by CPU & system performance*
6. **CPU Clocking**:
   a. Bộ xử lý bị quản lý bởi 1 clock có tốc độ ko đổi
   b. ***Clock Period*** (Clock Cycle **CC**) thời gian of 1chu kỳ clock.
   c. ***Clock Frequency*** (Clock Rate **CR**): số chu kỳ mỗi giây
      CC = 1/CR
      Vd: CC = 250ps = 250 x 10^{-12}s => CR = 1/CC = 4 GHz
7. **CPU execution time (CPU time)**

$$\frac{CPU\ execution\ time}{for\ a\ program} = \frac{\#CPU\ clock\ cycles\ for\ a\ program}{} \times clock\ cycle\ time$$

or

$$\frac{CPU\ execution\ time}{for\ a\ program} = \frac{\#CPU\ clock\ cycles\ for\ a\ program}{clock\ rate}$$

   Có thể tăng hiệu năng bằng cách giảm
   _ số lượng clock cycle cần thiết cho 1 chương trình
   _ hay độ dài của 1 clock cycle
8. VD: **Tính hiệu năng**: Chạy program trên máy tính A 2GHz clock trong 10s. Clock rate of program B bao nhiêu để nó chạy program này trong 6s; biết B cần 1,2 clock hơn máy tính A để chạy program
   Công thức số chu kỳ: **ClockCycles = CPUTime x ClockRate**
   Giải: Số chu kỳ của A:
   $ClockCycle_A = CPUTime_A \times ClockRate_A = 10s \times 2GHz = 20 \times 10^9$
   $ClockRate_B = 1.2 \times ClockCycle_A / CPUTime_B = 4GHz$
9. **Clock Cycles per Instruction CPI**: (số chu kỳ clock của 1 lệnh)

$$\frac{CPU\ Clock\ Cycles}{for\ a\ program} = \frac{Instruction\ Count}{for\ a\ program} \times \frac{Clock\ Cycles}{per\ Instruction}$$

và

$$CPUTime = InstructionCount \times CPI \times CC = I \times CPI/CR$$

   a. Ko phải tất cả các lệnh đều có cùng thời gian thực thi lệnh
   b. CPI là trung bình số chu kỳ clock mỗi lệnh cần để thực thi nó
10. VD: **So sánh hiệu năng**: Máy tính A&B hiện thực cùng 1 ISA. A có CC=250ps, CPI=2.0; B có CC=500ps, CPI=1.2. Ai nhanh hơn?
    Giải: Thời gian chạy chương trình (I Instruction Count) của A & B:
    $CPUTime_A = I \times CPI_A \times CC_A = I \times 2.0 \times 250ps = I \times 500\ ps$
    $CPUTime_B = I \times CPI_B \times CC_B = I \times 1.2 \times 500ps = I \times 600\ ps$
    => A nhanh hơn 600/500=1.2 lần B
11. Ví dụ: **CPI trung bình**: có 3 lớp lệnh A, B, C

Class	A	B	C	IC	CC	Avg.CPI
CPI for class	1	2	3			
IC in sequence 1	2	1	2	**2+1+2=5**	**2x1+1x2+2x3=10**	**10/5=2.0**
IC in sequence 2	4	1	1	**4+1+1=6**	**4x1+1x2+1x3=9**	**9/6=1.5**

    IC: Instruction Count
    Avg.CPI (CPI trung bình) ko kết luận nhanh chậm
12. **Chi tiết CPI**:
    a. Nếu lớp câu lệnh khác nhau thì số lượng chu kỳ cũng khác nhau.

$$ClockCycles = \sum_{i=0}^{n}(CPI_i \times InstructionCount_i)$$

    _ IC_i: số lượng (phần trăm) lệnh của class i thực thi
    _ CPI_i: số trung bình clock cycles mỗi lệnh của instruction class
    _ n: số lượng instruction class

$$CPI = \frac{ClockCycles}{InstructionCount} = \sum_{i=0}^{n}(CPI_i \times \frac{InstructionCount_i}{InstructionCount})$$

13. VD: **CPI của các class**

OP	**Freq**	**CPI_i**	FreqxCPI$_i$	(a)	(b)	(c)
ALU	50%	1	0.5	0.5	0.5	0.25
Load	20%	5	1.0	0.4	1.0	1.0
Store	10%	3	0.3	0.3	0.3	0.3
Branch	20%	2	0.4	0.4	0.2	0.4
CPI trung bình			2.2	1.6	2.0	1.95

    a. Dữ liệu đc cache nên giảm Load còn CPI_{Load}=2 chu kỳ
       CPU_{new}=1.6 x IC x CC; so với lúc đầu thì nhanh hơn 2.2/1.6 lần
    b. Giảm 1 cycle cho lệnh rẻ nhánh => CPI_{Branch}=1=>nhanh hơn 2.2/2
    c. 2 lệnh ALU có thể thực thi cùng lúc => nhanh hơn 2.2/1.95 lần
14. **Các yếu tố ảnh hưởng đến InstructionCount, CPI, Clock Cycle**:

	Instruction Count	CPI	Clock Cycle
Algorithm	**X**	X	
Programming language	**X**	X	
Compiler	**X**	**X**	
ISA	**X**	**X**	**X**
Core organization		**X**	**X**
Technology			**X**

15. **Power & Energy**:
    a. ***Dynamic power***: CMOS
    $Power_{dynamic}$=1/2 x CapacitiveLoad x $Voltage^2$ x FrequencySwitched
    b. ***Mobile devices***:
    $Energy_{dynamic}$=CapacitiveLoad x $Voltage^2$
    c. Fixed task, giảm clock rate (frequency switched), giảm power nhưng ko giảm energy.
16. **Reducing Power**:
    a. CPU mới có:
    _ 85% capacitive (điện dung) load của CPU cũ
    _ Giảm 15% voltage & 15% frequency

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}^2 \times F_{old}} = 0.85^4 = 0.52$$

    CPU mới dùng ít năng lượng hơn CPU cũ
    b. ***Bức tường năng lượng*** (***power wall***): chúng ta ko thể giảm voltage hay nhiệt thêm nữa
    c. ***Static Power***: vì rò rỉ xuất hiện ngay cả khi transitor tắt:

$$Power_{static} = Current_{static} \times Voltage$$

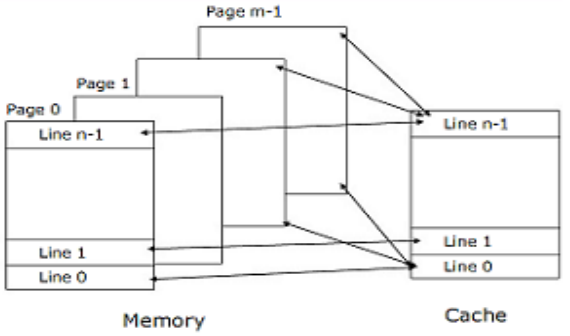
    _ Rò rỉ tăng lên trong bộ xử lý khi kích thước transistor nhỏ hơn
    _ Tăng số lượng transistor=> tăng tiêu thụ năng lượng ngay cả khi chúng bị tắt.
    _ ***ExecutionTimeRation*** = ***SPEC_ration*** = ***RefTime/ExTime***
    _ Geometric mean: $GM = \sqrt[n]{\prod_{i=1}^{n} ExecutionTimeRation_i}$
    ExecutiveTime=CPUTime=IC x CPI x CC
17. **SPEC Power Benchmark**: mức độ tiêu thụ năng lượng khác nhau ở các mức workload khác nhau.
    _ Performance: ssj_ops/sec
    _ Power: Watts (Joules/sec)
    => $Overall\ ssj_{opsper}\ Watt = (\sum_{i=0}^{10} ssj\_ops_i)/(\sum_{i=0}^{10} power_i)$
18. **Time maket**:
    _ Product life = 2W, đỉnh ở W
    _ Ontime = ½ * 2W * W
    _ Delayed = ½ * (W-D+W)*(W-D)
    _ Phần trăm lợi nhuận bị mất là (D(3W-D)/)

CACHE

1. **Thời gian truy xuất bộ nhớ trung bình:**
AverageAccessTime ≈ P_{hit} x T_{cache_access} + (1-P_{hit}) x (T_{mm_access}+ T_{cache_access}) x BlockSize + $T_{checking}$
   _ P_{hit} = khả năng cache hit; tỉ lệ cache hit
   _ T_{cache_access}= cache access time
   _ T_{mm_access}= main memory access time
   _ BlockSize = số lượng word trong cache block
   _ $T_{checking}$ = thời gian cần thiếc cho việc kiểm tra cache hit or miss
   VD: A computer has 8MB MM với 100ns access time, 8KB cache with 10ns access time, BS=4; $T_{checking}$=0; P_{hit}=0.97 => **AAT=22.9 ns**

2. **Các kỹ thuật tổ chức cache**
   - Ánh xạ trực tiếp (Đơn giản, nhanh – ánh xạ cố định)
   - Ánh xạ liên kết đầy đủ (Phức tạp, châm – ánh xạ linh hoạt)
   - Ánh xạ liên kết tập hợp (phức tạp nhanh – ánh xạ linh hoạt)

3. **Ánh xạ trực tiếp**



Tag	**Line**	**Word**

   Tag: là địa chỉ của trang trong bộ nhớ
   Line: là địa chỉ của line trong cache
   Word: là địa chỉ của word trong line
   Ví dụ: Bộ nhớ 4gb, cache 1mb, kick thước line 32 byte
   => line = 32 byte = 2^5 => word = 5 bit
   Cache = 1mb = 2^10 => có 2^10/2^4 line => line = 5 bit
   4 gb = 2^22 byte => tag = 22 – 10 = 12 bit
4. **Ánh xạ liên kết đầy đủ**

Tag	**Word**

   Ví dụ: bộ nhớ 4gb, cache 1mb, line 32 byte
   Line = 32byte = 2^4 => word = 5 bit
   Tag = 22 – 5 = 17 bit
5. **Ánh xạ liên kết theo bộ**
   Cache: được chia thành k way
   Bộ nhớ: Được chia thành m trang, mỗi trang có kích thước bằng kích thước way trong cache
   Ánh xạ: Mỗi page trong bộ nhớ sẽ được ánh xạ đến một way bất kỳ.

Tag	**Set**	**Word**

   Ví dụ: bộ nhớ 4gb, cache 2mb, 2 way, kích thước line 32 bytes
   Line size = 32 = 2^5 => word = 5 bit
   Cache = 1mb = 2^10 => có 2^10/2^5/2 = 2^4 line trong 1 way => set = 4 bit
   Tag = 22 – 5 – 4 = 13 bit
6. **Các chiến lược thay thế cache**
   - Thay thế ngẫu nhiên
   - FIFO

- LrU

7. Thay thế ngẫu nhiên
- Các block được chọn ngẫu nhiên để thay
- Đơn giản
- Tỷ lệ miss cao vì không xet tới block nào đang được sử dụng

8. Chiến lược FIFO
- Các block đoc vào cache trước sẽ được thay thế trước
- Tỷ lệ mis thập hơn ngẫu nhien nhưng vẫn cao vì không xét đến block nào đang thực sự sử dụng.

9. Chiến lược LRU
- Các block ít được sử dụng trong thời gian gần đây sẽ được chọn để thay
- Tỷ lệ miss thấp nhất.

10. Hiệu năng cache
- $\mathbf{T}_{access}$ = (Hitcost) + (miss rate) * (miss penaty)

11. Các yếu tố ảnh hưởng đến hiệu năng
- Kích thước cache
- Phân chia cache: dữ liệu – lệnh
- Tạo nhiều mức cache

12. Các kiểu miss
- **Compusory miss**: không tìm thấy bắt buộc
- **Capacity miss:** không đủ dung lượng
- **conflict miss**: Xung đột

13. Write through & Write back

14. AMAT = hit time + miss time * miss penalty
15. AMAT = % instructions * AMAT – I + %instructions * AMAT –D

VITUAL MEMORY

1. AMAT = $T_{mem} + (1+h).T_{disk}$

2. TBL : Translation Lookaside Buffer

3. Lý do sử dụng vitual memory
- Share main memory
- Simplify memory manage
- Provide protection

4. So sánh Cache và Vitual memory
- Chiến lược thay thế
  - Cache – điều khiển bởi HW
  - VM – điều khiển bởi OS

5. Replacement
- Orgnaztion : full associate
- Replacement : LRU
- Write policy : write back
- TBL

6. Segmentation
- Paging
- Segmentation
- Paging segmentation

7. Protection

8. VPN – PPN

Virtual Page Number VPN	Page offset
TLB	
Physical Page Number PPN	Page offset

MULTIPROCESSORS

1. Multiprocessors việc 2 processors cùng làm việc đồng thời tong một chương trình

2. Lợi ích:
- Giảm thiểu chi phí
- Tăng độ tin cậy
- Tăng băng thông

3. Flynn classification
- SÍD
- SIMD
- MISD
- MIMD

4. Memory consistency model
- Là cách thức mà bộ nhớ điều hành việc đọc và ghi của nhiều processors.
- Là tập hợp các quy tắc để 1 giá trị được ghi ở 1 processor này có thể được đọc ở một processor khác.

5. Đồng bộ hóa
- Bảo đảm các tiến trình đang chạy trên các processor có thể sử dụng chung 1 nguồn dữ liệu mà không bị conflit.
- Spin lock
- Barrier

6. Spin lock
- *l*ocks that a processor continuously tries to acquire, spinning around a loop until it succeeds.

7. Barrier
- Wait until all threads have reached a point in the program before any are allowed to proceed further
- Uses two shared variables
  - A counter that counts how many have arrived
  - flag that is set when the last processor arrives

8. Cache conhenrence
- Ensuare reading a location should return the lastest value written to that location.

Cấu trúc Pipelining

1. **CPU phải**: Fetch instructions; Interpret (dịch) instructions; Fetch data; Process data; Write data
2. **Instruction Cycle**: Fetch Cycle (slide 6/3); Indirect Cycle (slide 7/3); Execute Cycle (slide 8/3); Interrupt Cycle (slide 9/3)
3. **Pipeling Strategy**:
   a. ***Prefetch***:
      _ Fetch truy cập bộ nhớ chính
      _ Execution thường ko truy cập bộ nhớ chính
      _ Có thể fetch lệnh kế tiếp trong suốt quá trình thực thi lệnh hiện tại
      _ Gọi instruction prefetch
   b. ***Improved Performance***:
      _ Nếu ko cần phải doubled: fetch thường ngắn hơn execution; prefetch cần nhiều hơn 1 instruction. Bất cứ jump hay branch có nghĩa là các prefetched instructions ko đòi hỏi instructions
      _ Thêm trạng thái để tăng hiệu suất
4. **2-stage Instruction Pipeline**: slide 13/3
5. **Instruction Execution Stages**: tuần tự thực thi lệnh thông thường là:
   _ Fetch Instruction (***FI***)
   _ Decode Instruction (***DI***) xác định op-code & giá trị cụ thể toán hạng
   _ Calculate Operands (***CO***) tính toán địa chỉ hiệu dụng
   _ Fetch Operands (***FO***)
   _ Execute Instruction (***EI***) thực thi operation
   _ Write Operand (***WO***) lưu trữ kết quả vào bộ nhớ
6. **Pipeline Hazards**:
   a. Có nhiều tình huống làm lệnh kế tiếp trong dòng lệnh ko thể thực thi trong suốt clock cycle. Lệnh đc gọi là bị ***stalled***.
   b. ***Khi 1 instruction bị stalled***:
      _ Tất cả các lệnh sau stalled instruction trong pipeline đều bị stalled
      _ Ko có lệnh mới nào đc fetch trong khi stall xảy ra
   c. ***Các dạng of hazards (conflicts)***:
      _ ***Structural hazards***:
        - Hardware conflicts gây ra bởi vì dùng cùng nguồn tài nguyên phần cứng tại cùng thời điểm (vd memory conflicts …)
        - Nhìn chung, các tài nguyên phần cứng bị đụng độ sẽ đc duplicated để tránh structural hazards.
        - Các đơn vị chức năng (ALU, FP unit) cũng có thể là pipeline để hỗ trợ một vài instructions tài cùng thời điểm
        - Memory conflicts có thể đc giải quyết bằng cách:
          + Có 2 caches tách biệt, 1 cho instruction & 1 cho operands (Harvard architecture)
          + Dùng multiple banks của bộ nhớ chính hay duy trì kết quả trung gian bên trong các thanh ghi
      _ ***Data hazards***:
        - Gây ra bởi sự nghịch đảo thứ tự các operations phụ thuộc data do pipeline. (vd Write/Read conflicts, …)
        - Penalty do data hazards có thể bị giảm bởi 1 kỹ thuật đc gọi là ***forwarding*** (***bypassing***).
        - Kết quả ALU đc fed back đến ALU input. Nếu hardware xác định giá trị cần thiết cho operation hiện tại cho current operation. Giá trị này đc tạo từ previous operation (nhưng chưa đc viết vào). Nó chọn kết quả trước đó, thay vì giá trị từ thanh ghi & bộ nhớ
      _ ***Control hazards***: gây ra bởi các lệnh rẽ nhánh mà thay đổi thứ tự thực thi lệnh
7. **6-stage CPU Instruction Pipeline**: slide 26/3
8. **Number of Pipline Stages**: số lượng trạng thái lớn cho hiệu suất cao
   _ Số lượng trạng thái tăng overhead khi chuyển thông tin giữa những trạng thái & đồng bộ giữa các trạng thái
   _ Độ phức tạp của CPU tăng theo số lượng của trạng thái
   _ Khó để duy trì nhiều pipeline với tốc độ cao vì pipeline hazards
9. **Dealing withc Branches**:
   _ 1 trong những vấn đề chính là thiết kế instruction pipeline đảm bảo 1 dòng các instructions để khởi tạo các trạng thái of pipeline
   _ Khi lệnh thực thi, quan trọng là xác định branch thực thi hay ko
   _ Nhiều cách tiếp cận ≠ có thể xem xét với điều kiện rẽ nhánh: Wait, Multiple streams, Prefetch branch target, Loop buffer, Delayed branch, Branch prediction.
10. **Branch Handling**:
    a. ***Wait***: chờ pipeline cho đến khi lệnh branch thực thi đến last stage. Mất nhiều hiệu suất, từ 20%-35% các lệnh thực thi là branch (conditional & unconditional)
    b. ***Multiple Streams***: hiện thực bằng hardware để giải quyết các trường hợp có thể xảy ra.
       _ Có 2 ống pipeline
       _ Prefetch mỗi nhánh vào 1 ống pipeline tách biệt
       _ Dùng ống pipeline thích hợp
       _ Dẫn đến tranh chấp bus & register
       _ Đa nhánh đưa đến cần nhiều ống pipelines
    c. ***Pre-fetch branch target***: khi 1 điều kiện rẽ nhánh tìm thấy, mục tiêu of rẽ nhánh là pre-fetched, thêm instruction theo sau nhánh. Giữ target cho đến khi branch đc thực thi hay dùng ***IBM 360/91***.
    d. ***Loop Buffer***:
       _ Dùng bộ nhớ có tốc độ rất cao & nhỏ để giữ n fetched instructions gần đây nhất xuất hiện trong sequence.
       _ Nếu 1 nhánh đã phát hiện, buffer đc kiểm tra trước để xem nếu nhánh kết quả có phải là nó ko. Lệnh kế đc fetch từ buffer
       _ Bằng cách dùng pre-fetching, loop buffer sẽ chứa vài instruction tuần tự trước instruction hiện tại.
       _ Nếu 1 branch là kết quả có vị trí địa chỉ ở trước lệnh branch, kết quả sẽ sẵn sàng trong buffer. Hữu hiệu cho If-Then, If-Then-Else
       _ Kỹ thuật này là hoàn toàn phù hợp với loops hay sự lặp đi lặp lại. Các lệnh trong 1 loop đc fetch từ memory chỉ 1 lần
    e. ***Delayed Branch***:
       _ Tái sắp xếp các lệnh để rẽ nhánh xuất hiện sau
       _ Compiler tìm các lệnh mà có thể duy chuyển từ vị trí gốc đến vị trí chờ đợi & sẽ thực thi ko quan tâm đến kết quả rẽ nhánh (60%-85% thành công)
11. **Branch Prediction**:
    _ Khi tìm thấy rẽ nhánh, dự đoán đc tạo ra & theo đg dự đoán
    _ Các lệnh trên đg dự đoán đc fetched. Nhánh dự đoán dựa vào thông tin thống kê tĩnh hay động.
12. **Static Branch Prediction**: giả sử jump luôn xảy ra, luôn luôn fetch lệnh đích. Một vài lệnh rất giống kết quả trong phép jump hơn những cái còn lại. Có thể đạt thành công đến 75%.
13. **Dynamic Branch Prediction**: good for loops
    _ Dựa vào previous history
    _ Lưu trữ thông tin đối với những nhánh trong bản lịch sử rẽ nhánh, khi đó dự đoán chính xác nhánh đầu ra
    _ 1 hay 1 vài bits (history bits) có thể kết hợp với mỗi lệnh điều kiện rẽ nhánh. Nó cho biết lịch sử rẽ nhánh hiện tại of instruction.
    _ History bits đc lưu trữ tạm trong bộ nhớ tốc độc cao