**Kiến trúc máy tính nâng cao**

Chương 1: **Kiến trúc máy tính**

1. **Chi phí mạch**:
2. **CPU**:
   1. Chức năng chính của CPU là thực thi các lệnh đc lưu trong bộ nhớ chính. 1 lệnh bảo CPU thực thi 1..n tác vụ cơ bản của nó.
   2. CPU bao gồm các thanh ghi & bus giao tiếp bên trong để truyền data giữa control unit CU, ALU, registers. CU là bộ giải mã các lệnh để thực thi & điều khiển các thành phần khác làm việc.
   3. CPU bảo gồm 2 thành phần chính:
      1. A Program control unit
      2. An Arithmetic & Logic Unit (ALU)
3. **Registers**:
   1. CPU cần vùng lưu trữ dữ liệu tạm khi làm việc
   2. Các đơn vị lưu trữ này đc gọi là thanh ghi registers.
   3. Chúng là thành phần có level cao I trong cấu trúc bộ nhớ fân cấp
   4. Số lượng & chức năng of các thanh ghi ≠ giửa các máy tính
   5. Thanh ghi có 2 nhiệm vụ chính:
      1. Thanh ghi người dùng thấy đc: đc dùng bởi máy & mã assembly của lập trình viên. Gồm:
         * Các thanh ghi dùng cho mục đích chung
         * Thanh ghi dữ liệu
         * Thanh ghi địa chỉ
         * Thanh ghi điều kiện
      2. Thanh ghi điểu khiển & trạng thái: dùng bởi CU để điều khiển CPU & hệ điều hành để thực thi chương trình.
4. **Machine Instructions**: CPU chỉ có thể thực thi mã máy dạng binary, gọi là những lệnh máy:
   1. ***Một machine instruction cụ thể có các thông tin sau***:
      1. Cái gì phải thực hiện xong
      2. Thao tác này áp dụng cho ai
      3. Nơi lưu kết quả trả về
      4. Địa chỉ lệnh kế tiếp
   2. ***4 dạng của machine instruction***:
      1. Lệnh số học & luận lý
      2. Lệnh chuyển dữ liệu từ bộ nhớ vào thanh ghi của CPU
      3. Điều khiển chương trình
      4. Giao tiếp với thiếc bị ngoại vi
5. **Instruction Set Design**: Thiết kế tập lệnh là quan trọng của hệ thống máy tính.
   1. Những vấn đề quan trọng nhất là:
      1. Bao nhiêu & loại tác vụ sẽ có; độ phức tạp của tác vụ
      2. Kiểu dữ liệu
      3. Định dạng của lệnh: độ dài, số lượng các địa chỉ, kích thước của các trường khác nhau, …
      4. Số lượng các thanh ghi
      5. Mô hình đánh địa chỉ
   2. Các vấn đề này có quan hệ với nhau
6. **Machine Cycles** (chu kỳ máy)
   1. Thực thi một lệnh được thực hiện trong 1 chu kỳ máy
   2. CPU thực thi 1 lệnh sau lệnh khác, lập đi lập lại
   3. Thời gian chu kỳ máy cho biết hiệu năng (speed)
   4. Các lệnh khác nhau cần thời gian khác nhau để thực thi, thời gian thực thi trung bình đc dùng để tính toán hiệu năng
7. **Đánh giá hiệu năng CPU:**
   1. 2 thước đo là:
      1. **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***
      2. **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.
   2. ***Response time & Throughput bị ảnh hưởng bởi***:
      1. *Thay đổi CPU nhanh hơn*: tăng Response time & Throughput
      2. *Tăng số lượng CPU*: tăng Throughput
8. **Hiệu năng Performance (Speed)**:
   1. Hiệu năng lớn khi thời gian thực thi nhỏ nhất:
   2. Nếu X chạy nhanh gấp n lần Y thì
   3. Giảm thời gian đáp ứng (response time) hầu như luôn luôn cải thiện thông lượng (throughput).
9. **Đo hiệu năng**:
   1. ***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
   2. ***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
   3. *Different programs are effected differently by CPU & system performance*
10. **CPU Clocking**:
    1. Bộ xử lý bị quản lý bởi 1 clock có tốc độ ko đổi
    2. ***Clock Period*** (Clock Cycle **CC**) thời gian of 1chu kỳ clock.
    3. ***Clock Frequency*** (Clock Rate **CR**): số chu kỳ mỗi giây

CC = 1/CR

Vd: CC = 250ps = 250 x 10-12s => CR = 1/CC = 4 GHz

1. **CPU execution time** (**CPU time**)

or

Có thể tăng hiệu năng bằng cách giảm

* + 1. số lượng clock cycle cần thiếc cho 1 chương trình
    2. hay độ dài của 1 clock cycle

1. 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:

ClockCycleA=CPUTimeA x ClockRateA=10s x 2GHz=20x109

ClockRateB=1.2xClockCycleA / CPUTimeB = 4GHz

1. **Clock Cycles per Instruction CPI**: (số chu kỳ clock của 1 lệnh)

và

* 1. Ko phải tất cả các lệnh đều có cùng thời gian thực thi lệnh
  2. CPI là trung bình số chu kỳ clock mỗi lệnh cần để thực thi nó

1. 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:

CPUTimeA= I x CPIA x CCA = I x 2.0 x 250ps = I x 500 ps

CPUTimeB= I x CPIB x CCB = I x 1.2 x 500ps = I x 600 ps

=> A nhanh hơn 600/500=1.2 lần B

1. 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

1. **Chi tiết CPI**:
   1. Nếu lớp câu lệnh khác nhau thì số lượng chu kỳ cũng khác nhau.
      1. ICi: số lượng (phần trăm) lệnh của class i thực thi
      2. CPIi: số trung bình clock cycles mỗi lệnh của instruction class
      3. n: số lượng instruction class
2. VD: **CPI của các class**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **OP** | **Freq** | **CPIi** | FreqxCPIi | (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 |

* 1. Dữ liệu đc cache nên giảm Load còn CPILoad=2 chu kỳ

CPUnew=1.6 x IC x CC; so với lúc đầu thì nhanh hơn 2.2/1.6 lần

* 1. Giảm 1 cycle cho lệnh rẻ nhánh => CPIBranch=1=>nhanh hơn 2.2/2
  2. 2 lệnh ALU có thể thực thi cùng lúc => nhanh hơn 2.2/1.95 lần

1. **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** |

1. **Power & Energy**:
   1. ***Dynamic power***: CMOS

Powerdynamic=1/2 x CapacitiveLoad x Voltage2 x FrequencySwitched

* 1. ***Mobile devices***:

Energydynamic=CapacitiveLoad x Voltage2

* 1. Fixed task, giảm clock rate (frequency switched), giảm power nhưng ko giảm energy.

1. **Reducing Power**:
   1. CPU mới có:
      1. 85% capacitive (điện dung) load của CPU cũ
      2. Giảm 15% voltage & 15% frequency

CPU mới dùng ít năng lượng hơn CPU cũ

* 1. ***Bức tường năng lượng*** (***power wall***): chúng ta ko thể giảm voltage hay nhiệt thêm nữa
  2. ***Static Power***: vì rò rỉ xuất hiện ngay cả khi transitor tắt:
     1. Rò rỉ tăng lên trong bộ xử lý khi kích thước transistor nhỏ hơn
     2. Tăng số lượng transistor=> tăng tiêu thụ năng lượng ngay cả khi chúng bị tắt.
     3. ***ExecutionTimeRation*** = ***SPEC\_ration*** = ***RefTime/ExTime***
     4. Geometric mean:

ExecutiveTime=CPUTime=IC x CPI x CC

1. **SPEC Power Benchmark**: mức độ tiêu thụ năng lượng khác nhau ở các mức workload khác nhau.
   * 1. Performance: ssj\_ops/sec
     2. Power: Watts (Joules/sec)

=>

1. **Time maket**:
   * 1. Product life = 2W, đỉnh ở W
     2. Ontime = ½ \* 2W \* W
     3. Delayed = ½ \* (W-D+W)\*(W-D)
     4. Phần trăm lợi nhuận bị mất là (D(3W-D)/)
2. **Computer design must meet**:
   * 1. Functional requirements
     2. Area/performance/cost/power goals: tối ưu, ước lượng, tìm kiến trúc khả thi nhất
     3. Cân nhắc các nhân tố khác: thời gian ra sản phẩm, định hướng công nghệ, …
3. **Speeding it up**:
   * 1. Pipelining
     2. On board cache; on board L1 & L2 cache
     3. Dự đán lệnh rẽ nhánh
     4. Phân tích dòng dữ liệu
     5. Thực hiện theo phỏng đoán
4. **Performance Balance**:
   * 1. Tăng tốc bộ xử lý
     2. Tăng dung lượng bộ nhớ
     3. Memory speed lags behind processor speed
5. **Thu hẹp khoảng cách tốc độ bộ nhớ & CPU**:
   * 1. Tăng số lượng bit truy xuất trong 1 lần đọc/ghi. Tạo DRAM “wider” (rộng) hơn là “deeper” (sâu)
     2. Change DRAM interface
     3. Reduce frequency of memory access. More complex cache & cache on chip
     4. Increase interconnection bandwidth:
        + High speed buses
        + Hierarchy of buses

Chương 2: **Hệ thống bộ nhớ**

1. **Memory characteristics**:
   1. ***Location***:
      1. Internal: lưu các dữ liệu đang được CPU dùng
      2. External: vùng nhớ lâu dài & lớn; lưu trử data & progam. Truy xuất qua I/O controllsers
   2. ***Capacity*** (sức chứa)
      1. *Word size*: số lượng bit dùng biểu diễn 1 byte; =độ dài câu lệnh
      2. *Number of words* (or bytes)
   3. ***Unit of transfer***:
      1. Các phần tử có thể đc định địa chỉ
      2. Đơn vị giao tiếp: số lượng bit có thể đọc/ghi bộ nhớ trong 1 lần; ko nên bằng 1 word hay 1 addressable unit.
      3. Internal: thường bị chi phối bởi độ rộng của bus
      4. External: thường 1 block lớn hơn 1 word.
   4. ***Access method***:
      1. *Sequential* (tape): bộ nhớ đc tổ chức thành các đơn vị data (records), bắt đầu ở điểm đầu & đọc tuần tự. Thời gian truy xuất dựa vào vị trí của data & vị trí trước đó.
      2. *Derect* (disk): các block độc lập có địa chỉ duy nhất; truy cập bằng cách nhảy tuần tự tìm kiếm. Thời gian truy xuất phụ thuộc vào vị trí data & vị trí trước đó.
      3. *Random* (Ram) Individual addresses identify locations exactly; Access time is independent of location or previous access.
      4. *Associative* (Cache) Data is located by a comparison with contents of a portion of the store. Access time is independent of location or previous access.
   5. ***Performance***:
      1. *Thời gian truy xuất* (Access time):
         * Của RAM là từ khi có địa chỉ đến khi data sẳn sang
         * For non-RAM, it’s the time between positioning the read/write mechanism at the desired location
      2. *Memory cycle time*:
         * Thời gian cần cho việc “phục hồi” trước lần truy xuất kế
         * **Cycle time = Access + Recovery**
      3. *Transfer Rate*: tốc độ mà data vào ra 1 memory unit
         * For RAM: **transdfer rate = 1/(cycle time)**
         * For non-RAM: **TN = TA + N/R**
           1. TN: thời gian trung bình để đọc ghi N bits
           2. TA: thời gian trung bình để truy xuất
           3. N: số lượng bit
           4. R: tốc độ truyền, đơn vị bps
   6. ***Physical type***: Semiconductor (RAM0, Magnetic (Disk & Tape), Optical (CD, DVD), Others (Buble, Hologram)
   7. ***Physical characteristics***:
      1. Decay:
      2. Volatility: tính bay hơi
      3. Erasable: xóa đc
      4. Power consumption: tiêu thụ năng lượng
   8. ***Organization***: Cách sắp xếp những bít vào word, ko phải lúc nào nó cũng rỏ rang. Giống như xếp sách
2. **Thuộc tính quan trọng của bộ nhớ**:
   1. ***Speed***: tốc độ càng nhanh càng tốt
   2. ***Size***: kích thước càng lớn càng tốt
   3. ***Cost***: Chi phí hợp lý

=> Chúng ta phải xác định công nghệ dùng để hiện thực bộ nhớ

1. **Main memory**:
   1. MM có thể lưu như 1 tập hợp các tế bào nhớ có thể nhớ 1 word.
   2. Mỗi cell đc gán 1 địa chỉ duy nhất & địa chỉ này đc đánh số 0…n
   3. Số lượng bit đại chỉ xác định kích thước của bộ nhớ (16 bits-65K, 24 bits-16M; 32 bits-4G)
   4. ***Memory control unit*** has a read/write control mechanism & an address selection mechanism.
2. **Memory Bandwidth**: biểu diển số lượng data có thể truy xuất từ bộ nhớ trong 1 giây.

M=Bandwidth = AmountOfDataPerAccess / MemoryCycleTime

* 1. *Vd*: MCT=100ns & 4 bytes (a word) per access

=>M-bandwidth = 40MBps

* 1. *Có 2 kỹ thuật cơ bản để tăng bandwidth của 1 bộ nhớ*:
     1. Giảm memory cycle time: nhưng đắc & size bộ nhớ bị giới hạn
     2. Chia bộ nhớ thành nhiều bank,mỗi bank chứa control unit riêng

1. **Memory Hierarchy**:
   1. *Phân cấp*:
      1. Registers: trong CPU
      2. Internal or Main memory: bao gồm nhiều cấp cache; RAM
      3. External memory: Backing store
   2. Khi duyệt bộ nhớ phân cấp từ trong CPU ra ngoài thì:
      1. Giảm chi phí/bit (cost/bit)
      2. Tăng dung lượng
      3. Tăng thời gian truy xuất
      4. Giảm mật độ truy xuất bởi CPU.
2. **Thời gian truy xuất bộ nhớ trung bình**:

AverageAccessTime ≈ Phit x Tcache\_access + (1-Phit) x (Tmm\_access+ Tcache\_access) x BlockSize + Tchecking

* + 1. Phit = khả năng cache hit; tỉ lệ cache hit
    2. Tcache\_access= cache access time
    3. Tmm\_access= main memory access time
    4. BlockSize = số lượng word trong cache block
    5. Tchecking = 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; Tchecking=0; Phit=0.97 => **AAT=22.9 ns**

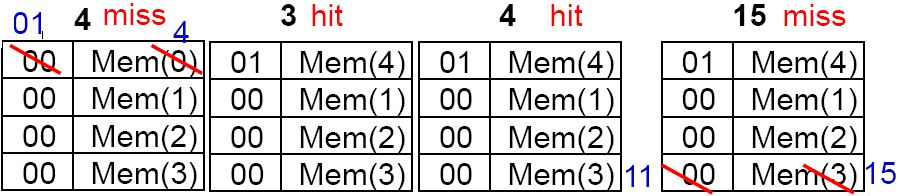
1. **Thuật ngữ trong Memory Hierachy**:
   1. **Hit**: data ở block trong upper level memory
      1. *HitRate*: là tốc độ truy xuất upper để phát hiện data ở đó
      2. *HitTime*: thời gian truy xuất upper level memory; bao gồm RAM access time + Time to determine hit/miss
   2. **Miss**: data ko ở trong upper mà ở trong lower level memory:
      1. *MissRate* = 1 – HitRate
      2. *MissPenalty*: Time thay thế 1 block trong upper + Time chuyển block vào Processor
      3. HitTime ≪ MissPenalty
2. **Cache Design**: kích thước &số block đc copy fải đc thiết kế cẩn thận, cũng như g/thuật quyết định bỏ block nào ra khỏi cache khi cache đầy
   * 1. Cache block size
     2. Total cache size
     3. Mapping function
     4. Replacement method
     5. Write policy
     6. Number of caches: single, 2 or 3 level; Unified >< Split cache
3. **Mapping Function**:
   1. Cần để ánh xạ bộ nhớ chính vào 1 dòng cache & xác định khối bộ nhớ chính nào đang đc gắn với 1 cache line
   2. Chọn hàm ánh xạ là xác định cách tổ chức cache
   3. Ex:
      1. Cache 64Kbytes; CacheBlock=4bytes =>có 16k (214) block line
      2. Data are transferred in blocks of 4byte. 16MByte main memory

=> 4M block, mỗi block 4 byte

1. **Direct Mapping Cache**: mỗi block of bộ nhớ chính đc ánh xạ vào 1 cache slot cố định. Vì vậy một số phần tử ở lower level phải chia sẽ vị trí với upper level.
   1. ***Công thức***: i = j modulo m
      1. i = cache line number
      2. j = main memory block number
      3. m = number of lines in the cache.
   2. ***Cache line***: đc đánh thứ tự ***Index***. Chia thành 3 phần:
      1. ***Valid***: xác nhận data trong cache line có hợp lệ hay ko
      2. ***Tag***: đc tính bằng công thức ***tag = i = j modulo m***
         * i = Số thứ tự của cache line
         * j = số lượng block trong bộ nhớ chính
         * m = số lượng line trong cache
      3. ***Data***
   3. ***Địa chỉ***: độ dài là **S+W**, đc chia làm 2 phần:
      1. W bit xác định duy nhất 1 word (hay byte offset) trong 1 block of bộ nhớ chính. Word (hay byte) này trong block sẽ đc dùng.
      2. S bit xác định 1 khối bộ nhớ (bộ nhớ chính có 2S block)
         * A cache line field: R bit; chứa giá trị ***index*** của cache line
         * A tag: S-R bits; chứa giá trị modulo với m.
      3. VD: 24 bits địa chỉ, trong đó có 2 bit xác định word (4 byte block), 22 bit xác định block (8 bit tag & 14 bit slot or line)

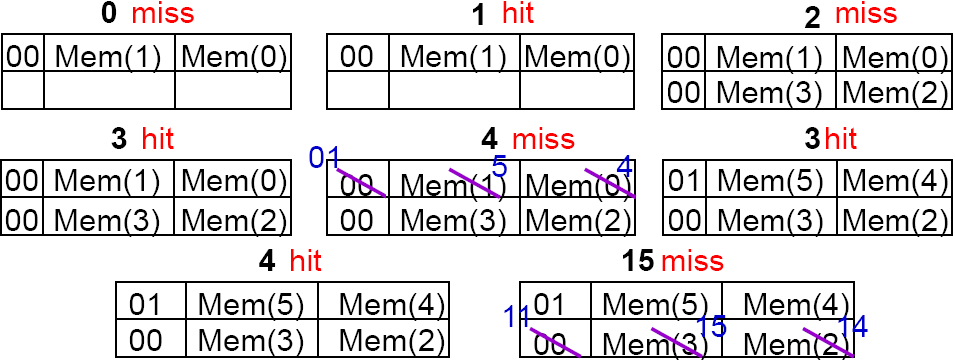
|  |  |  |
| --- | --- | --- |
| Tag (**S-R** bits): 8 | Line/Slot/Index (**R** bits): 14 | Word (**W** bits): 2 |

* + - * Ko có 2 block trong cùng 1 dòng có cùng Tag field.
      * Kiểm tra nội dung cache bằng cách tìm line & kiểm tra Tag.
  1. ***So trùng***: CPU phân tích Address:
     1. Từ giá trị ***Index*** của Address, CPU lấy cache line tương ứng
     2. Nếu cache line đó ***Valid*** & ***TagAddress=TagCacheLine*** thì Cache line này đc chọn;
     3. Vì BlockSize = LineSize nên trả về ***Data*** = CacheLine.
  2. ***Pros & Cons***:
     1. Hiện thực đơn giản nên ko đắc tiền.
     2. Luân phiên truy xuất 2 block đc ánh xạ vào cùng 1 cache slot sẽ làm cache miss rate tăng cao => gọi là hiện tượng ***thrashing***.
  3. ***Summary***:
     1. Độ dài địa chỉ: (S+W) bits
     2. Số lượng phần tử có thể đánh địa chỉ = 2S+W words hay bytes
     3. BlockSize = LineSize = 2W words or bytes
     4. Số lượng Block trong bộ nhớ chính = 2S+W/2W = 2S
     5. Số lượng Line trong cache = m = 2R
     6. Kích thước của tag = (S-R) bits



4 = **01**00 15 = **11**11

1. **Multiword Block Direct Mapped Cache**:
   1. Giống Direct Mapped Chache nhưng Data của cache line gồm nhiều Word. Khi đó trường Word (Byte offset) của địa chỉ chứa index của Word cần đọc ra.
   2. Do đó, khi ***Valid*** & ***Tag*** hợp lệ, W bits của trường ***Word*** trong Address sẽ xác định Word cần truy xuất; trả về cho Data



* + 1. Đọc 1 lần 2 ô nhớ Mem(i+1), Mem(i) vào cache block.
    2. Tag = i div 2W; W là số bit của phần Word. Vd: 15 = **11**11

1. **Associative Mapping**:
   1. 1 block bộ nhớ chính có thể đc load vào bất kỳ slot nào của cache
   2. Memory Address = Tag + Word
   3. Tag xác định Block of Memory duy nhất
   4. Để xác định nếu 1 block trong cache ko, 1 thiếc bị cần để kiểm tra mỗi slot’s tag => mất thời gian cho quá trình tìm kiếm
   5. ***Ex: Address structure & Cache Line Structure***:
      1. *Address*: 22 bit tag lưu trữ block data dài 32bits=4byte=2word

|  |  |
| --- | --- |
| Tag: 22 bit | Word: 2 bit |

* + 1. *Cache line*: 16Kline Cache

|  |  |
| --- | --- |
| Tag: 22 bit | Data: 32 bit |

* + 1. So sánh TagAddress & TagCacheLine để kiểm tra *hit*.
    2. 2 bit of address xác định 16 bit word trong 32 bit data block
    3. Ex:
       - Data 32 bits có giá trị là FEDCBA98
       - Address 16339C
       - Tag = Address div 2W = 058CE7
  1. ***Summary***:
     1. Address length = (S+W) bits
     2. Số lượng phần tử có thể đánh địa chỉ: 2S+W words/bytes
     3. BlockSize = LineSize = 2W words/bytes
     4. Số lượng blocks trong bộ nhớ chính = 2S+W/2W=2S
     5. Số lượng line trong cache ko xác định đc
     6. Kích thước Tag = S bits

1. **Set Associative Mapping**: Miss Rate tỉ lệ nghịch số tập hợp v
   1. Cache đc chia thành v tập hợp; mỗi tập hợp chứa k slots:

m = k \* v

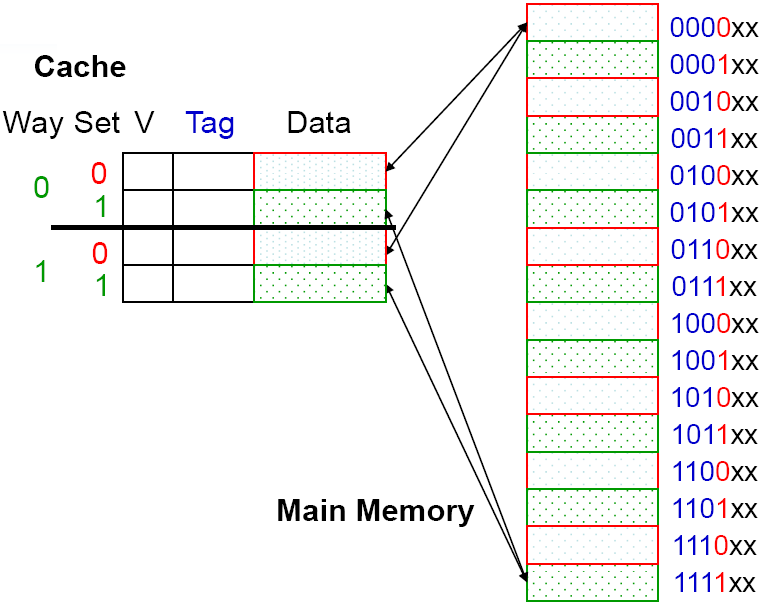
* 1. Công thức xác định Cache Set: **i = j modulo v**. Với block j có thể là bất kỳ trong Cache Set i.

*Ex*: 2 slot mỗi set (k=2); block có thể là 1 trong 2 block của set

* 1. *Direct mapping*: ***k=1***; *Full associative*: ***v=1*** (k = số slot of cache)
  2. *Ex*: 13 bit set number =>Số block trong bộ nhớ chính modulo 213.

Các block 000000, 008000, 00A000, 00C000,…,FF8000 ánh xạ vào cùng 1 Cache Set 0.

* 1. ***Ex: Address structure & Cache Line Structure***: v =2
     1. Address: Tag + Set + Word
        + *Tag*: (s-d) bit; các bit cao xác định TagAddress
        + *Set*: d = lg(v) = 1 bit xác định cache set. SetNumber=AddressNumber div
        + *Word*: w = 32 bit thấp nhất định nghĩa word blocks
     2. So trùng tất cả TagCache trong 1 Cache Set với TagAddress để xác định memory block có trong cache hay ko (hit/miss)



* 1. *Ex: Tag 9 bit; Set 13 bit; Word 2 bit*
     1. Address = 167FFC;
     2. Tag = 02C (9 bit đầu)
     3. SetNumber = 1FFF (13 bit kế)
  2. ***Summary***:
     1. Độ dài địa chỉ = (s+w) bits
     2. Số lượng đơn vị có thể đánh địa chỉ: 2S+W words/bytes
     3. BlockSize=LineSize=2W words/bytes
     4. Số lượng blocks trong bộ nhớ chính: 2S+W/2W=2S
     5. Số lượng line trong 1 tập hợp = k
     6. Số lượng tập hợp = v = 2d
     7. Số lượng line trong cache = k.v = k. 2d
     8. Kích thước của tag = (s-d) bits

1. **Replacement Algorithms** (RA)
   1. Với direct mapping thì ko cần RA
   2. Với associative mapping, RA cần để xác định block cần replace:
      1. Dùng phần cứng để hiện thực giải thuật => tăng tốc độ
      2. Least-recently used (***LRU***) thay block ở trong cache mà lâu nhất ko đc dùng đến
      3. First in first out (***FIFO***)
      4. Least-frequently used (***LFU***) thay block ít đc tham khảo nhất
      5. Random
2. **Write Policy**:
   1. ***Vấn đề***:
      1. Duy trì sự nhất quán giửa cache content & main memory content với chi phí performance thấp
      2. Ko overwrite cache block nếu memory of nó vẫn chưa cập nhật
      3. I/O có thể định địa chỉ trực tiếp bộ nhớ chính
      4. Nhiều CPU phải có các caches độc lập nhau
   2. ***Write hit***:
      1. *Cho phép cache & memory ko giống nhau* (***write-back***)
         * Ghi data của cache block này chỉ khi nó bị thay thế bởi block khác (ghi vào next level trong memory hierarchy)
         * Cần ***dirty*** bit cho mỗi cache block để biết nếu nó cần ghi ra memory khi bị “đuổi ra” khỏi cache ko
      2. *Bắt buộc cache memory giống nhau*:
         * Luôn luôn ghi data cả cache block & next level in the memory hierarchy (***write-through***) => ko cần ***dirty*** bit
         * Ghi vào next level sẽ chậm => dùng ***write buffer***, chỉ khi buffer đầy thì mới ghi
   3. ***Write Buffer for Write-Through Caching***:
      1. *Mô hình*:

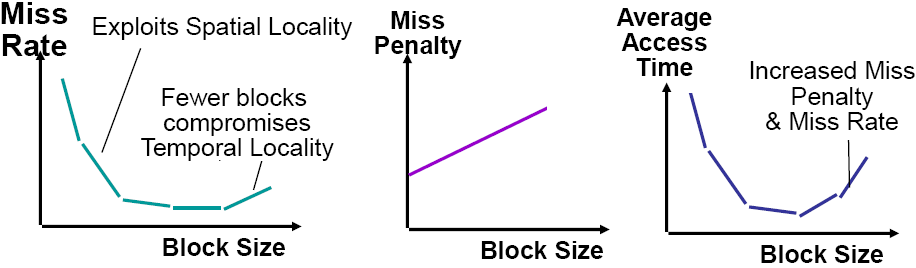
|  |  |
| --- | --- |
|  | * + - * Processor: ghi data vào cache & main memory       * Memory Controller: ghi dữ liệu của buffer vào memory |

* + 1. Write buffer chỉ là 1 FIFO; hoạt động tốt nếu

StoreFrequency (w.r.t time) ≪ 1/DRAM write cycle

* + 1. Vấn đề write buffer là StoreFrequency 🡪1/DRAM write cycle làm write buffer trở nên bảo hòa (saturation). Giải pháp là dùng 1 write-back cache; hay dùng L2 cache

1. **Cources of Cache Misses**:
   1. ***Compulsory***
   2. ***Confilict***: nhiều ô nhớ ánh xạ vào cùng 1 vị trí cache. Giải pháp: Tăng cache size hay tăng associativity.
   3. ***Capacity***: cache ko thể có tất cả các block của 1 chương trình => tăng cache size
2. **Cache Coherence**:
   1. **Software Solution**:
      1. Dựa vào phân tích code: xác định code ko an toàn khi caching, đánh dấu chúng để biết ko cached chúng => thêm code để làm điều này
      2. Compiler & OS thỏa thuận để giải quyết điều này
      3. Tăng Compile Time; fức tạp khi truyền từ hardware →software
      4. Giải pháp này thích hợp với các quyết định thận trọng
   2. **Hardware Solution**:
      1. Cache coherence protocols
      2. Dynamic recognition of potential problems
      3. Run time
      4. More efficient use of cache
      5. Transparent to programmer
      6. Two main categories:
         * ***Directory protocols***:
           1. lưu trữ & duy trì thông tin về copy của data trong cache.
           2. Directory lưu trong main memory.
           3. Yêu cầu kiểm tra directory
           4. Dy chuyển dữ liệu phù hợp sẽ đc thực thi: giửa memory & cache hay giữa các cache
           5. Tạo thắc cổ chai ở trung tâm
           6. Hiệu quả với scale system lớn với lược đồ kết nối bên trong phức tạp
         * ***Snoopy protocols***: phân phát cache coherence cho các cache controllers. Cache nhận ra 1 line đc shared; update sẽ cảnh báo các caches khác. Tăng bus traffic. Phù hợp với bus của multiprocessor
3. **Miss Rate, Block Size & Cache Size**:



AverageMemoryAccessTime = HitTime + MissPenalty x MissRate

1. **Do hiệu năng của Cache**:
   1. 2 kỹ thuật để cải thiện hiệu năng cache:
      1. Giảm missrate bằng cách giảm khả năng 2 memory block khác nhau cùng chiếm 1 vùng
      2. Giàm miss penalty bằng cách tăng thêm level của hierarchy
   2. Hiệu năng:

CPUTime = IC x CPI x CC = IC x CPIstall x CC

CPIstall = CPIideal + MemoryStallCycles

MemoryStallCycles = MissRate x MissPenalty

ReadStallCycles = Reads/Program x ReadMissRate x ReadMissPenalty

WriteStallCycles=(Writes/Program x WriteMissRate x WriteMissPenalty) + WriteBufferStalls

* 1. *Vd*: CPIideal = 1.1; 50% arith/logic, 30% ld/st, 20% control. 10% data memory miss with 50 cycle miss penalty

CPI = CPIideal+AverageStallsPerInstruction

=1.1(cycle) + (0.30(datamemops/instr) x 0.10(mis/datamemop) x 50(cycle/miss) )

= 1.1 cycle + 1.5 cycle = 2.6 cycle

* + 1. Vậy 1.5/2.6=58% thời gian CPU bị stalled chờ truy xuất bộ nhớ
    2. 1% miss rate làm CPI tăng 1.5/10 = 0.5.
  1. *Vd*: CPIideal=2; 100 cycle miss penalty; 36% load/store instr’s, 2% I$; 4% D$ miss rate.

MemoryStallCycles = 2% x 100 + 36% x 4% x 100 = 3.44

CPIstall = 2 + 3.44 = 5.44

1. **Reducing Cache Miss Rates**:
   1. *Cho phép các block ánh xạ linh động hơn*: fully associative cache hay n-way set associative.
   2. *Dùng nhiều mức cache*: CPIideal=2; 100 cycle miss penalty (to main memory); 36% load/store, 2% L1$ & 4% D$ miss rate, UL2$ (cho lệnh & data) có 25 cycle miss benalty & 0.5% miss rate.

CPIStalls= 2 + .02x25 + 0.36x.04x25 + 0.05x100 + .36x0.05x100 = 3.54

CPIStalls\_NoL2= 2 + 0.02x100 + .36x.04x100 = 5.44

1. **Vấn đề cân nhắc khi thiết kế Cache nhiều mức**:
   1. ***Thiết kế caches L1 & L2 khác nhau***:
      1. Cache đầu tiên nên tập trung giảm đến mức ít nhất ***hit time*** để hổ trợ shorter clock cycle. Smaller with smaller block sizes
      2. Cache thứ 2 nên tập trung giảm ***miss rate*** để giảm penalty thời gian truy xuất bộ nhớ chính. Larger with larger block sizes.
   2. Miss penalty của L1 cache đc giảm xuống bởi có mặt của L2 cache. Do đó, L1 có thể smaller & faster nhưng miss rate cao
   3. L2 cache thì hit time ít quan trọng hơn miss rate:
      1. L2$ hit time xác định bởi L1$ miss penalty
      2. L2$ local miss rate ≫ global miss rate
2. **Improving Cache Performance**: Slide 125-126
   1. *Giảm hit time của cache*: Cache nhỏ hơn; Cache ánh xạ trực tiếp; Blocks nhỏ hơn; for writes
   2. *Giả miss rate*: cache lớn hơn; chiến lược ánh xạ linh động; các block lớn hơn; Victim cache (small buffer holding most recently discarded blocks)
   3. *Reduce the miss penalty*: smaller blocks, dùng write buffer …
3. **RAM**: truy xuất ngẩu nhiên, Read/.Write; bay hơi; lưu trử tạm thời; tĩnh hay động.
4. **Transistor**:
   1. nMos: tích cực dương (dẩn khi gate = +)
   2. pMos: tích cực âm (dẩn khi gate = -)
5. **Lưu trữ byte Syndrome word**:
   1. Mỗi bit của syndrome là 0|1; nếu bit i =1 thì bit datai lổi
   2. K bit syndrome word: giá trị từ 0..2K-1
      1. 0 nếu ko có lổi đc phát hiện
      2. 2K-1 giá trị xác định bit bị lổi
   3. Lổi có thể bị phát hiện trong M data bits hay K check bits
      1. 2K - 1 ≥ M+K
      2. Vd: 1 word 8 data bits (M=8), K ≥ 4
6. **Characteristics of Syndrome**:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Position | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
|  | 1100 | 1011 | 1010 | 1001 | 1000 | 0111 | 0110 | 0101 | 0100 | 0011 | 0010 | 0001 |
| Data bit | D8 | D7 | D6 | D5 |  | D4 | D3 | D2 |  | D1 |  |  |
| Check bit |  |  |  |  | C8 |  |  |  | C4 |  | C2 | C1 |

Chapter 3: **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**:
   1. ***Prefetch***:
      1. Fetch truy cập bộ nhớ chính
      2. Execution thường ko truy cập bộ nhớ chính
      3. Có thể fetch lệnh kế tiếp trong suốt quá trình thực thi lệnh hiện tại
      4. Gọi instruction prefetch
   2. ***Improved Performance***:
      1. 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
      2. 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:
   * 1. Fetch Instruction (***FI***)
     2. Decode Instruction (***DI***) xác định op-code & giá trị cụ thể toán hạng
     3. Calculate Operands (***CO***) tính toán địa chỉ hiệu dụng
     4. Fetch Operands (***FO***)
     5. Execute Instruction (***EI***) thực thi operation
     6. Write Operand (***WO***) lưu trữ kết quả vào bộ nhớ
6. **Pipeline Hazards**:
   1. 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***.
   2. ***Khi 1 instruction bị stalled***:
      1. Tất cả các lệnh sau stalled instruction trong pipeline đều bị stalled
      2. Ko có lệnh mới nào đc fetch trong khi stall xảy ra
   3. ***Các dạng of hazards (conflicts)***:
      1. ***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 hổ trợ một vài instructions tài cùng thời điểm
         * Memory conflicts có thể đc giải quyết bằng cách:
           1. Có 2 caches tách biệt, 1 cho instruction & 1 cho operands (Harvard architecture)
           2. Dùng multiple banks của bộ nhớ chính hay duy trì kết quả trung gian bên trong các thanh ghi
      2. ***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
      3. ***Control hazards***: gây ra bở 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
   * 1. Số lượng trạng thái tăng overhead khi chuyển thông tin giữa những trạng thái & đồng bộ giữa các trạng thái
     2. Độ phức tạp của CPU tăng theo số lượng của trạng thái
     3. Khó để duy trì nhiều pipeline với tốc độ cao vì pipeline hazards
9. **Dealing withc Branches**:
   * 1. 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
     2. Khi lệnh thực thi, quan trọng là xác định branch thực thi hay ko
     3. 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**:
    1. ***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)
    2. ***Multiple Streams***: hiện thực bằng hardware để giải quyết các trường hợp có thể xảy ra.
       1. Có 2 ống pipeline
       2. Prefetch mỗi nhánh vào 1 ống pipeline tách biệt
       3. Dùng ống pipeline thích hợp
       4. Dẩn đến tranh chấp bus & register
       5. Đa nhánh đưa đến cần nhiều ống pipelines
    3. ***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. Giử target cho đến khi branch đc thực thi hay dùng ***IBM 360/91***.
    4. ***Loop Buffer***:
       1. Dùng bộ nhớ có tốc độ rất cao & nhỏ để giử n fetched instructions gần đây nhất xuất hiện trong sequence.
       2. 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
       3. Bằng cách dùng pre-fetching, loop buffer sẽ chứa vài instruction tuần tự trước instruction hiện tại.
       4. 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
       5. 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
    5. ***Delayed Branch***:
       1. Tái sắp xếp các lệnh để rẽ nhánh xuất hiện sau
       2. 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**:
    * 1. Khi tìm thấy rẽ nhánh, dự đoán đc tạo ra & theo đg dự đoán
      2. 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
    * 1. Dựa vào previous history
      2. Lưu trữ thông tin đối với những nhánh trong bản lịch sử rẽ nhánh, khi đó dự đoán chính xác nhánh đầu ra
      3. 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.
      4. History bits đc lưu trữ tạm trong bộ nhớ tốc độc cao

Chương 4: **Reduced Instruction Set Computers (RISC)**

1. **RISC**:
   * 1. Thể hiện sự đổi mới quan trọng trong kiến trúc máy tính
     2. Là sự cố gắn để tạo thêm sức mạnh cho CPU bằng cách dùng tập lệnh máy đơn giản
     3. Đặc điểm chính:
        + Nhiều thanh ghi cho mục đích chung hay dùng kỹ thuật biên dịch để tối ưu việc dùng thanh ghi
        + Giới hạn & đơn giản tập lệnh
        + Nhấn mạnh vào việc tối ưu tập lệnh cho pipeline
2. **Complex Instruction Set Computer (CISC)**: ngược với RISC. Cả 2 kiến trúc RISC & CISC đc fát triển như cố gắng khác phục ~ lổ hỏng ngữ nghĩa (semantic gap), do đó giảm ~ chi phí phát triển software
3. **Semantic Gap**: để tăng hiệu quả of phát triển software, ~ ngôn ngữ lập trình mới & có sức mạnh lớn đc phát triển, cung cấp mức độ trừu tượng cao. Sự tiến hóa này làm tăng khoảng cách ngữ nghĩa giữa các ngôn ngữ lập trình & ngôn ngữ máy
4. **Ước lượng thực thi chương trình**:
   1. Chúng ta cần phải xác định các đặc tính thực thi of tập lệnh máy tuần từ (do chương trình viết bằng ngôn ngữ bậc cao sinh ra)
   2. Các khía cạnh quan tâm:
      1. Tần số thực thi của các operations
      2. Các kiểu operands & tần suất dùng chúng
      3. Thực thi tuần tự
      4. Điều khiển dòng: branches, loops, subprogram calls
5. **Ước lượng kết quả**:
   * 1. Tần suất thực thi các lệnh máy
     2. Các mô hình đánh địa chỉ:
        + Các lệnh chính dùng mô hình đánh địa chỉ đơn
        + Các mô hình đánh địa chỉ phức tạp (địa chỉ gián giếp, indexed + indirect, …) chỉ đc dùng ~18% các lệnh
6. **Operands**:
   * 1. Operand types:
        + 74-80% các operands là scalars (intergers, reals, characters, …)
        + 20-26% là array/structures; 90% của chúng là global variables
        + 80% của scalars là local variables
     2. Các operands chính là ~ biến cục bộ có kiểu scalar, chúng có thể lưu trong các registers
     3. Tối ưu nên tập trung vào lưu trữ & truy xuất các biến scalar cục bộ
7. **Procedure calls**:
   * 1. Xem xét phần trăm of tất cả thời gian thực thi & thời gian thực thi các phát biểu of ngôn ngữ bậc cao:
        + Hầu hết thời gian dùng cho việc thực thi lện Calls & Returns
        + Chỉ 15% phát biểu of ngôn ngữa bậc cao (Call hay Return) thực thi thì chúng cũng thực thi gần hết toàn thời gian (vì chúng phức tạp)
        + Call hay Return đc kết hợp vào quan hệ tuần tự của các lệnh máy với nhiều tham chiếu bộ nhớ
     2. Chi phối thời gian:
        + Phụ thuộc vào số tham số truyền vào: hầu hết lời gọi (98%) đc truyền với ít hơn 6 tham số; hầu hết các biến (92%) là local scalar
        + Phụ thuộc vào mức độ lòng nhau
8. **Các đặc điểm chính of RISC**:
   * 1. Số lượng các lệnh đơn giản là ít (bé hơn 100).
        + Yêu cầu decode & execution hardware đơn giản & nhỏ
        + Cần có hard-wired controller hơn là microprogramming
     2. Các đặc điểm chính:
        + Thực thi nhiều lệnh máy trong 1 chu kỳ xung clock
        + Register-to-Register operations
        + Các cơ chế đánh địa chỉ đơn giản
        + Simple instruction formats
9. **One Instruction per Clock Cycle**:
   * 1. Chu kỳ máy đc định nghĩa là thời gian fetch 2 operans từ các registers, thực thi 1 ALU operation & store kết quả vào 1 register
     2. Instruction pipeline thực thi nhiều hiệu quả hơn do simple instructions & similar execution patterns.
     3. ~ operations fức tạp đc thực thi = dãy tuần tự các simple instructions
     4. Vd: 80% chương trình là lệnh đơn giản, 20% là lệnh phức tạp. CISC có 4 chu kỳ cho lệnh đơn giản, 8 chu kỳ cho lệnh phức tạp, cycle time là 100ns; RISC là 1 cycles, 14 cycles & 75 ns. Nếu có 106 lệnh thì CISC thực thi tốn: (106x0.8x4+106x0.2x8)x10-7=0,48 s

RISC: (106x0.8x1+106x0.2x14)x0.75x10-7=0,27 s

1. **Register-to-Register Operation**:
   1. Kiến trúc Load-and-Store: chỉ các lệnh Load & Store tham khảo data trong memory. Tất cả các lệnh thực thi chỉ với các registers (là các lệnh register-to-register)
   2. Đặc tính này làm đơn giản tập lệnh, nên control unit cũng đơn giản
   3. Kiến trúc này khuyến khích tối ưu việc dùng thanh ghi, do đó tuần suất ngẩu nhiên các operands duy trì trong high-speed storage
   4. Có nhiều thanh ghi:
      1. Biến & kết quả trung gian có thể đc lưu trong registers & ko cần lặp đi lặp lại việc load/stored từ bộ nhớ
      2. Tất cả biến cục bộ of procedures & các tham số truyền vào có thể đc lưu trong registers
      3. Có nhiều registers là đặc điểm điển hình of RISC vì nó giảm độ phức tạp of processor, có nghĩa là chúng ta có nhiều ko gian silicon ở trên processor để hiện thực chúng
2. **Register Windows**:
   * 1. Nhiều thanh ghi thường rất có ích
     2. Nếu nội dung of các thanh ghi phải đc lưu mỗi khi gọi procedure, do đó nhiều register nghĩa là duy trì đc lâu hơn trong register
     3. Giải pháp cho vấn đề này là chia register file thành tập các cửa số có kích thước cố định
        + Mỗi cửa sổ thanh ghi đc ấn định cho 1 procedure
        + Các cửa sổ cho các procedures kế bị overlapped để cho phép truyền tham số
3. **Chỉ dùng đánh địa chỉ đơn giản**:
   * 1. Hầu như tất cả các lệnh RISC dùng đánh địa chỉ thanh ghi đơn giản
     2. Các mô hình phức tạp có thể tổng hợp trong software từ ~ thứ đơn giản hơn
4. **Lệnh có chiều dài & format cố định**:
   * 1. Loading & decoding of instructions đơn giản & nhanh; nó ko cần chờ cho đến khi độ dài of 1 lệnh đc biết để bắt đầu decoding nó
     2. Decoding đc làm đơn giản vì opcode & address fields đc xác định tại cùng 1 vị trí cho tất cả các instructions
5. **Thuận lợi chính of RISC**:
   * 1. Tối ưu kiến trúc: Frequently executed instructions, Memory reference, Procedure call/return; Pipeline design
     2. Thiết kế ít phức tạp, giảm chi phí sản xuất & giảm thời gian giữa thiết kế & đem ra thị trường
6. **Điểm yếu của RISC**:
   * 1. Một operation phải cần đến 2, 3 hay nhiều instructions để hoàn tất. Nhiều truy xuất bộ nhớ, tốc độ thực thi phải bị giảm xuống trong các ứng dụng cụ thể
     2. Nó dẩn đến chương trình dài hơn, cần n` ko gian bộ nhớ để lưu trữ
     3. Khó để lập trình mã máy & chương trình assembly
7. **RISC vs. CISC**:
   * 1. Nhiều so sánh hiệu suất cho thấy benchmark của RISC > CISC
     2. Khó để biết đặc điểm nào của RISC làm nó có hiệu suất cao
     3. “CISC fans” cho là tốc độ cao hơn ko phải do các đặc tính RISC cụ thể tạo ra mà do technology, better compilers, …
8. **Các đặc điểm chính of CISC**:
   * 1. Nhiều lệnh (>200) & các lệnh kiểu dữa liệu phức tạp
     2. Mô hình đánh địa chỉ nhiều & phức tạp
     3. Hiện thực trực tiếp bằng phần cứng cho các phát biểu HLL
     4. Microprogramming techniques đc dùng để kết hợp các instructions có thể đc hiện thực
     5. Nghẽn cổ chai là vấn đề chính, do các mô hình đánh địa chỉ phức tạp & đa truy xuất bọ nhớ trên mỗi lệnh
     6. Tập lệnh nhiều nên compiler có các lệnh gần với lệnh HLL
     7. Khi các chương trình nhỏ hơn về kích thước, chúng thực thi nhanh hơn. Do chúng ít tốn ko gian bộ nhớ, ít lệnh thực thi & cần ít chu kỳ máy, nên ít thời gian thực thi hơn.
     8. Hiệu quả thực thi of chương trình cũng cải thiện vì hiện thực các operation phức tạp trong microcode hơn là machine code.
9. **Problems with CISC**: slide 36-39/4

Chương 5: **Superscalar Processors**

1. **Superscalar Architecture**:
   1. Superscalar là 1 máy tính đc thiết kế để cải thiện hiệu suất thực thi of các scalar instructions.
   2. Trong superscalar architecture (**SSA**), 1 vài lệnh có thể khởi tạo đồng thời & thực thi độc lập
   3. Pipelining cho phép vài instruction đc thực thi tại cùng thời điểm, nhưn phải là khác các trạng thái ống pipeline tại cùng thời điểm
   4. SSA bao gồm tất cả các đặc điểm of pipelining nhưng thêm vào đó vài instruction thực thi đồng bộ trong cùng trạng thái pipeline
   5. Các SSA mở đầu cho bước song song mới gọi là ***instruction-level parallelism***.
2. **How to implement the idea**:
   1. SSA processor fetch nhiều instructions cùng lúc, & cố gắng tìm instructions gần kề độc lập với instruction khác mà có thể thực thi //
   2. Dựa vào phân tích độc lập, processor sẽ phát & thực thi các lệnh theo thứ tự khác với thứ tự mã máy ban đầu
   3. Processor phải loại ra các ràng buộc ko cần thiết bằng cách dùng thêm các thanh ghi & thay đổi các tham chiếu đến register
3. **Superscalar Concepts**:
   1. Các kiến trúc superscalar cho phép vài lệnh có thể đc phát & hoàn tất mỗi clock cycle
   2. Super scalar architecture có 1 số ống pipeline có thể hoạt động //
   3. Phụ thuộc vào số lượng & loại đơn vị //, 1 số instructions có thể đc thực thi //
4. **SuperPipelining**:
   1. SuperPipelining dựa vào việc chia nhỏ các trạng thái of pipeline thành nhiều trạng thái con, & do đó tăng số lệnh có thể đc thực thi trong pipeline tại cùng thời điểm
   2. Trong 1 kiến trúc cụ thể & tập lệnh tương ứng, tỉ số tối ưu pipeline stages/sub-stages tăng quá sẽ làm giảm hiệu suất tổng thể
5. **SuperPipelined Superscalar Design**: slide 13/5
6. **Các giớ hạn thực thi song song**:
   1. ***ILP*** nói đến mức độ mà ~ programs có thể thực thi // ở mức lệnh
   2. Tối ưu ILP cần fải kết hợp việc tối ưu compiler & kỹ thuật hardware
   3. Vấn đề cản trở instructions thực thi // bởi kiến trúc superscalar là rất giống với cản trở làm giảm hiệu xuât thực thi of kiến trúc pipelined (pipelined hazards)
      1. True data dependency
      2. Control (Procedural) dependency
      3. Resource conflicts
      4. Outpt dependency
      5. Anti-dependency
7. **Procedural Dependency**:
   * 1. Các lệnh rẽ nhánh tạo nên vấn đề chính để bảo đảm việc tối ưu //
     2. Nếu các lệnh có độ dài thay đổi => ko thể fetched & issued song song cho đến khi instruction đã đc giải mã để biết số fetch cần thiết
8. **Resource Conflicts**: vài instructions tranh chấp cùng tài nguyên phần cứng tại cùng thời điểm. Có thể giải quyết 1fần=cách thêm tài nguyên
9. **Data Conflicts**: gây ra bởi sự phụ thuộc data giữa instructions trong program. Xác định vấn đề & tăng mức độ thực thi //, SSA cung cấp quyền tự do lớn để instructions có thể phát & thực thi. Do đó, phụ thuộc data co thể đc giải quyết
10. **Out of Order Execution**: SSA khai thác khả năng // mức lệnh tìm tang trong program. Điều này đạt đc bằng ***dynamic instruction scheduling***. Kết quả // fải giống với kết quả khi thực thi tuần tự
11. **Window of Execution**:
    * 1. Do phụ thuộc data, chỉ vài phần of instructions là có thể thực thi //
      2. Để tìm ra instructions có thể phát //, processor phải chọn từ dãy lệnh tuần tự lớn đầy đủ.
      3. ***Window of Execution***: Tập hợp instructions đc cân nhắc thực thi tại 1 thời điểm. Bất kỳ instruction trong window có thể đc phát thực thi //, dựa vào phụ thuộc data & ràng buộc resource
      4. Số lượng các lệnh trong window nên lớn nhất có thể. Tuy nhiên số lượng fetch instructions ở tốc độ cao là bị giới hạn & vấn đề branches
      5. Window of Execution có thể giải quyết vấn đề ***branch prediction***.
12. **Data Dependencies**:
    1. Tất cả instructions trong window of execution phải bắt đầu thực thi nếu ko phụ thuộc data & ràng buộc tài nguyên
    2. 3 kiểu phụ thuộc dữ liệu có thê đc xác định là:
       1. True data dependency (flow dependency/***write-read*** dependency)
       2. Output dependency (***write-write*** dependency)
       3. Anti- dependency (***read-write*** dependency)

**Artificial** (giả tạo/nhân tạo) **dependency**. Chúng ko phải là true data dependency. Chúng có thể đc giải quyết bằng cách gọi ***register renaming***.

1. **Instruction-level parallelism (ILP)**: Số lượng trung bình các lệnh trong chương trình mà processor có thể thực thi trong cùng thời điểm. Đc xác định bằng số true data dependencies & procedural (control) dependencies lien quan đến số instructions
2. **Machine parallelism of a processor**: khả năng processor đạt đc ILP of chương trình. Xác định số lượng instructions có thể fetched & executed tại cùng thời điểm. Số lượng hardware & tốc độ & cơ chế mà processor dùng để xác định các instructions độc lập.

Để đạt đc hiệu suất cao, chúng ta cần ILP & manchine parallelism

1. **Instruction Issue & Completion**: thực thi lệnh song2 có thể phân loại dựa trên 3 hoạt động:
   1. ***Instruction issue***: 1 lệnh đc khởi tạo & bắt đầu thực thi
   2. ***Instruction completion***: 1 lệnh hoàn tất operations cụ thể of nó
   3. ***Instruction commit***: ~ kết quả of instruction operations đc ghi vào register files hay cache. (machine state is changed)
2. **Instruction Execution Polocies**:

Instructions sẽ đc thực thi theo thứ tự khác với thứ tự ban đầu, nhưng vẫn cho kết quả chính xác

* 1. ***IOI withc IOC***: slide 38-39/5
  2. ***IOI with OOC***: slide 42/5
  3. ***OOI withc OOC***: slide 43-44/5

1. **Superscalar Implementation**:
   * 1. Fetch đồng thời multiple instructions
     2. Logic xác định true dependencies liên quan đến các giá trị register
     3. Cơ chế để truyền đạt các giá trị
     4. Cơ chế khở tạo multiple instructions in parallel
     5. Tài nguyên cho thưc thi song song đa lệnh
     6. Cơ chế cho việc commiting process state theo đúng thứ tự

Chương 6: **Very Long Instruction Word (VLIW) Processors**

1. **Superscalar**: kiến trúc phức tạp, cần nhiều hardware for run-time detection, giới hạn về how far we can go witch this technique. Instruction window for execution bị giới hạn kích thước => giới hạn khả năng // instructions
2. **VLIW Processors**: vài operations có thể đc thực thi // đc đặt vào cùng 1 single instruction word.
   * 1. Các kiến trúc VLIW đc // hóa tại thời điểm biên dịch
     2. Khi 1 instruction đc fetched, tất cả các operations tương ứng đc thực phát //. Ko cần phần cứng phát hiện // tại thời điểm thực thi
     3. Instruction window problem đc giải quyết. Compiler có thể phân tích toàn bộ chương trình để xác định parallel operations
3. **Explicit Parallelism**:
   * 1. Song song tại thời điểm biên dịch. Processor dùng nó để thực thi //
     2. Yêu cầu phần cứng ít phức tạp hơn. Số lượng FUs có thể tăng mà ko cần phần cứng phức tạp để xác định // như superscalars
     3. Compiler có nhiều thời gian để xác định parallel operations
     4. Good compilers có thể xác định // dựa vào fân tích tổng thể program

=> ***Explicit Parallel Instruction Computing*** (***EPIC***)

1. **Main Problems**:
   1. Số lượng các thanh ghi cần để duy trì hoạt động of FUs (lưu trữ operands & kết quả)
   2. Lượng lớn data truyền nhận là cần giữa các FUs & register files & giữa register files & memory
   3. Băng thong rộng giữa instructions cache & fetch unit
   4. Large code size, 1 phần của operations ko dùng đến gây lãng phí
   5. Ko tương thích với mã nhị phân
2. **Loop unrolling**: slide 12-17/6

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mem R | MemR | FP1 | FP2 | I/BRA |

* + 1. 2 tham khảo bộ nhớ, 2 FP operations & 1 interger operation hay rẽ nhánh có thể đc đặt trong 1 instruction
    2. Delay cho 1 double word load là 1 clock cycle
    3. Delay cho floating point operation là 2 clock cycles
    4. Ko có clock cycles cho integer operations
    5. ***Loop Unrolling***: là 1 kỹ thuật dùng compilers để tăng khả năng // của program. Nó cho phép sinh ra code hiệu quả hơn cho các processors với mức độ song song lệnh cao hơn

1. **Loop Unrolling in General**:
   * 1. Cho 1 tập resources cố định (kiến trúc processor) & 1 vòng lặp, có 1 giới hạn về số lần lặp nên unrolled.
     2. Loop unrolling tăng kích thước bộ nhớ cần cho việc lưu trữ program
     3. Good compiler phải tìm cách tối ưu level of unrolling for each loop
     4. Hardware support cần để giử VLIW processor busy:
        + Nhiều registers để lưu nhiều operations đang active in parallel
        + Nhiều traffic phải đc hổ trợ trong parallel: giữa register files & memory; giữa register files & functional units

Chương 7: **Parallel Processing**

1. **Tính toán tuần tự**:
   * 1. Hầu hết ngôn ngữ lập trình máy tính yêu cầu lập trình viên chỉ rỏ giải thuật như là các instructions tuần tự
     2. Các bộ xử lý thực thi chương trình bằng các lệnh máy một cách tuần tự & 1 lệnh tại mỗi thời điểm
     3. Mỗi lệnh đc thực thi bởi các bộ xử lý tuần tự (fetch instruction, fetch operands, perform operation, store results)
2. **Cải thiện hiệu xuất**:
   * 1. Giảm thời gian thực thi của instruction
        + Tăng chu kỳ xung clock nhanh hơn
        + Tập lệnh đơn giản (RISC)
     2. Song song trong processor:
        + ***Pipelining***: song song cấp giai đoạn lệnh
        + ***Parallel execution of instructions (ILP)*** song song cấp lệnh, gồm Superscalar, VLIW architectures
     3. Song song processing: xử lý // nhằm tăng h/suất, tốn ít thời gian
3. **Tại sao là Prarallel Processing**?
   * 1. Máy tính tuần tự thường ko đạt thể đc hiệu suất cần thiết trong các ứng dụng cụ thể
     2. Các ứng dụng này thường có lượng tính toán rất lớn & khối lượng input data rất lớn
     3. Để cung cấp đủ hiệu suất cho các ứng dụng, chúng ta có thể có nhiều processors trong 1 máy tính đơn
4. **Parallel Computer**:
   * 1. Parallel Computer thích hợp với các kiến trúc có nhiều CPUs chạy song song để thực thi 1 ứng dụng cụ thể.
     2. Máy tính có thể đc cấu trúc bằng ~ cách ≠, dựa vào các đặc tính key:
        + Số lượng & độ phúc tạp of các CPUs riêng lẻ
        + Hiện hữu bộ nhớ chia sẻ
        + Topology kết nối bên trong (interconnection topology)
        + Hiệu suất of mạng nội kết nối (interconnection network)
        + I/O devices
5. **Parallel Programs**:
   * 1. Để giải quyết vấn đề dùng máy tính song song, 1 máy phân rả vấn đề thành các vấn đề nhỏ, chúng có thể thực thi song song
     2. Vì dữ liệu độc lập giữa các sub-problems, ko dễ dàng để phân chia vấn đề phức tạp để tạo ra sự song song mức độ cao
     3. Vì độc lập dữ liệu, các processors phải giao tiếp với nhau
     4. Thời gian cho việc giao tiếp thường rất cao hơn so processing time
     5. Communication scheme phải lập lịch tốt để có 1 giải thuật // tốt
6. **Flynn’s Classification of Architectures**:
   * 1. Phân loại Flynn (1966) dựa vào bản chất của luồng lệnh thực thi bởi máy tính & của luồng data mà các lệnh đưa đến
     2. Vô số dòng lệnh & dòng dữ liệu cho chúng ta 4 phân loại:
        + ***Single instruction, Single data stream – SISD***: 1 bộ xử lý, 1 dòng lệnh, data lưu trử trong bộ nhớ đơn, bộ xử lý đơn. (**Uniprocessor**)
        + ***Single instruction, Multiple data stream – SIMD***:
          1. Luồng lệnh máy tính đơn, sự thực thi đồng thời trên tập dữ liệu ≠
          2. SIMD với ***Shared Memory*** hay với ***Distributed Memory***.
          3. Có nhiều thành phần xử lý (processing elements), có bước Lockstep đồng bộ giửa các thành phần xử lý. Các thành phần xử lý có thể có bộ nhớ dữ liệu riêng hay chia sẻ bộ nhớ dùng chung thông qua mạng kết nối bên trong
          4. ***Array processor*** & ***Vector processors*** là ~ máy SIMD phổ biến I
        + ***Multiple instruction, Single data stream – MISD***:
          1. Data đơn tuần tự truyền đến 1 tập các processors
          2. Mỗi processor thực thi lệnh tuần tự
          3. Ko bao giờ đc hiện thực
        + ***Multiple instruction, Multiple data stream – MIMD***:
          1. Tập hợp các processor cho mục đích chung
          2. Thực thi đồng thời các lệnh tuần tự khác nhau, tập data khác nhau
          3. Phân lớp MIMD có thể chia nhỏ hơn nữa:
        + *Chia sẽ bộ nhớ*: Các bộ xử lý chia sẻ bộ nhớ & giao tiếp thong qua bộ nhớ chia sẽ. ***Symmetric multiprocessor*** (***SMP***) chia sẻ bộ nhớ đơn (share single memory or pool), chia sẻ bus để truy xuất memory; thời gian truy cập bộ nhớ là ngang nhau với các bộ xử lý. ***Non-uniform memory access*** (***NUMA***) thời gian truy xuất bộ nhớ của các bộ xử lý có thể khác nhau.
        + *Bộ nhớ phân tán* (loosely coupled) = ***Clusters***: tập hợp các uniprocessors độc lập hay SMPs. Kết nối bên trong tạo nên 1 dạng of cluster. Giao tiếp qua các đg cố định or mạng kết nối
7. **Performance of Parallel Architectures**:
   1. ***Important questions***:
      1. Khả năng máy tính // chạy nhanh đến mức nào?
      2. Chúng ta mong đợi máy tính // thực thi ứng dụng cho trước nhanh đến mức nào?
      3. Cách chúng ta đo hiệu suất of máy tính // & cải thiện hiệu suất?
   2. ***Performance metrics***:
      1. *Tốc độ đỉnh* (***Peak rate***): tốc độ tính toán lớn nhất có thể đạt đc trên lý thuyết, khi tất cả module dùng hết. Nó ko có ý nghĩa thực tế với người dùng, chỉ dùng cho nhà sản xuất trên thị trường
      2. *Speedup*: đo lường độ lợi bằng cách dùng máy tính // để giải quyết bài toán // cụ thể. S = TS/TP. Với Ts là thời gian thực thi of giải thuật tuần tự, Tp là thời gian thực thi of giải thuật song song
      3. *Efficiency* (hiệu suất): liên quan đến speedup & số processor đã dùng; E = S/P. Với S là speedup; P là số lượng processors. Trên lý thuyết (ideal situation) S=P với E=1. E ko thể đạt đến 1.
   3. ***Amdahl’s Law***:
      1. Tìm mức tối đa cải thiện của toàn hệ thống khi chỉ cải thiện 1 phần hệ thống. Thường dùng tính toán song song để dự đoán speedup tối đa (về mặt lý thuyết) khi dùng đa processors.
      2. Speedup of 1 chương trình dùng đa processors khi tính toán // bị giới hạn bằng thời gian cần cho phần nhỏ tuần tự of chương trình
      3. *Ex*: 95% program có thể //, speedup lý thuyết tối đa tính toán // là 1/(1-95%)=20x
      4. f là tỉ lệ tính toán (tùy giải thuật) fải thực hiện tuần tự (0≤f≤1); P là số processor
      5. Để đạt đc hiệu quả cao khi dùng nhiều processor, f phải nhỏ (giải thuật phải // cao), khi đó:
   4. ***Other Aspects that Limit the Speedup***:
      1. Ngoài tuần tự of giải thuật, có nhiều nhân tố giới hạn speedup: chi phí giao tiếp; load balancing of processors; chi phí tạo & lập lịch processors; I/O operations.
      2. Có nhiều giải thuật với mức độ // cao: giá trị f rất nhỏ & có thể bỏ qua; phù hợp với hệ thống // lớn; ~ nhân tố ≠ như chi phí giao tiếp là rất quan trọng.
   5. ***Efficiency & Communication Cost***:
      1. Xem xét tính toán // mức độ cao, khi đó f có thể bỏ qua. Chúng ta định nghĩa fC là communication overhead of 1 processor:
         * Tcalc: time mà processor thực thi tính toán
         * T­comm: time mà processor chờ đợi trong giao tiếp

(nếu fC là nhỏ)

* + 1. Với ~ giải thuật có mức độ // cao, các máy tính // lớn, có nhiều bộ processor, hiệu quả khi fC là nhỏ. Time cho việc giao tiếp là rất nhỏ so với time tính toán.
    2. Để có đc fC nhỏ hợp lý, số lượng processes có thể bị giới hạn

1. **Interconnection Network**:
   1. **Interconnection Network** (IN)
      1. IN là thành phần quan trọng of kiến trúc. Nó đc quyết định dựa vào hiệu suất tổng thể & chi phí tổng thể of kiến trúc
      2. Traffic trong IN gồm data transfer & transfers of commands & requests (control information)
      3. Các tham số quan trọng of IN là: tổng băng thông (bits/second), chi phí hiện thực
   2. **Single Bus**:
      1. Những mạng bus đơn là đơn giản & rẻ
      2. 1 giao tiếp đơn cho phép tại 1 thời điểm; bandwidth đc chia sẽ bởi tất cả các nodes
      3. Hiệu suất thấp. Để giử hiệu suất, số node phải giới hạn (16-20)
   3. **Completely Connected Network**:
      1. Mỗi nodes kết nối với mọi nodes khác
      2. Giao tiếp có thể đc thực hiện // giửa các cặp nodes
      3. Cả hiệu suất & chi phí đều cao
      4. Chi phí tăng nhanh hơn số lượng node
   4. **Crossbar Network**:
      1. Mạng động: topology của kết nối bên trong có thể thay đổi bằng cách cấu hình lại các switches.
      2. Kết nối hoàn toàn: bất kỳ node có thể kết nối trực tiếp với nhau
      3. Ít kết nối bên trong hơn cần hơn mạng kết nối hoàn toàn tĩnh. Tuy nhiên, cần số lượng switches lớn
      4. Số lượng lớn giao tiếp có thể đc thực thi //. 1 node có thể nhận hay gởi 1 data tại 1 thời điểm.
   5. **Mesh Network**:
      1. Rẽ hơn ~ mạng liên thông hoàn toàn nhưng vẫn cung cấp hiệu suất tốt
      2. Để truyền data giữa 2 node, đg đi qua các nodes ở giửa là cần thiết (maximum 2\*(n-1) ở giửa trong lưới n\*n)
      3. Khả năng cung cấp wraparound connections
      4. Lưới 3 chiều cũng có thể đc hiện thực
   6. **Hypercude Network**: 2-D, 3-D, 4-D, 5-D
      1. 2n nodes đc sắp xếp trong ma trận n chiều. Mỗi node đc kết nối n node hàng xóm
      2. Để truyền data giữa 2 nodes, đường truyền giửa các nodes trung gian là cần thiết (maximum n trung gian)