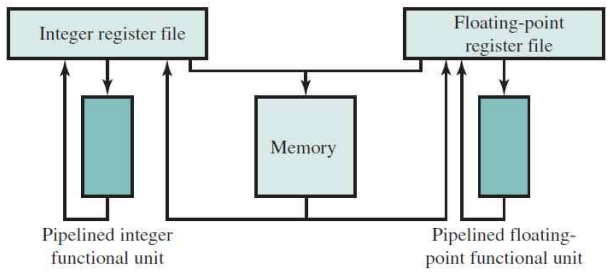
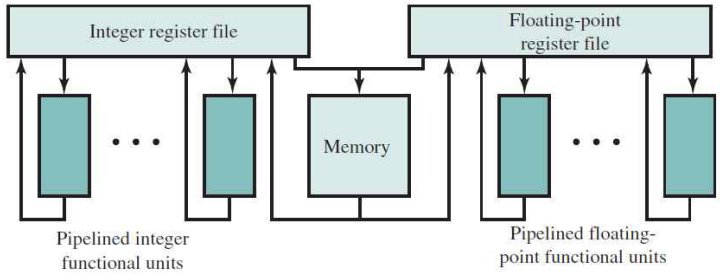
슈퍼스칼라 개요

1. Superscalar 프로세서

- **Superscalar 등장배경**
  - 1987년 scalar 명령어 실행속도 향상 방법으로 제안.

- **Superscalar processor**
  - 여러 개의 명령어들을 동시에 독립적으로 실행할 수 있는 프로세서
    - 여러 개의 명령어 실행 파이프라인을 사용
    - 명령어 실행순서 변경(out-of-order)까지 확장
  - 프로세서 성능향상을 위한 표준 방법
  - RISC와 CISR 모두 적용가능 : 주로 RISC

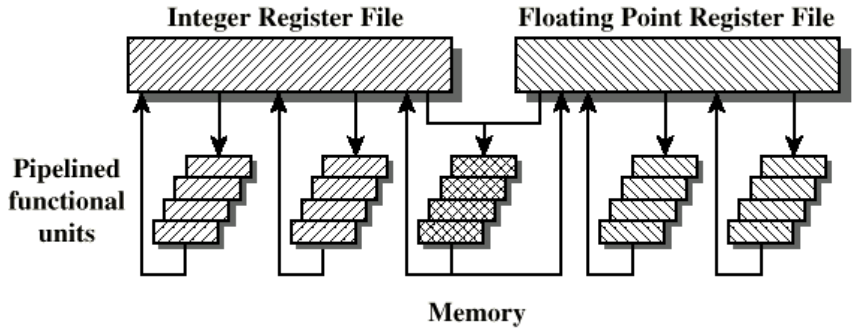




1. Superscalar 프로세서

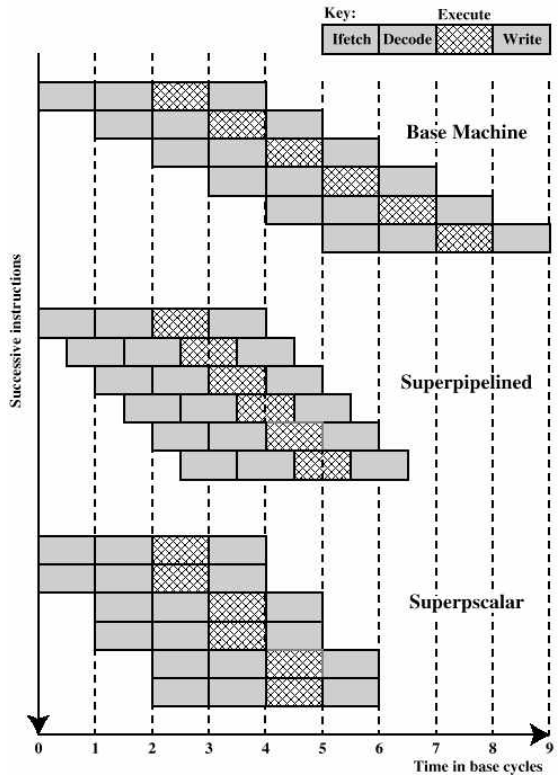
- **Superscalar 프로세서의 성능향상**
  - speedup factors : 1.58 ~ 8

- **스칼라 명령어 실행 예**
  - 여러 개의 명령어를 서로 다른 파이프라인을 사용해서 실행함으로써, 동시에 독립적으로 살행할 수 있다.
    - 2개의 정수연산, 2개의 부동소수점 연산, 1번의 메모리 액세스를 동시에 실시가능.



1. Superscalar 프로세서

- **Superascalar vs. superpipeline**



2. Superscalar 제한 요소

- **슈퍼스칼라 구조는 기본적으로 상호 독립적인 여러 개의 명령어들을 동시에 실행하는 것.**
  - 프로그램에서 동시에 실행할 수 있는 정도를 명령어 수준의 병렬성(ILP) 라고 한다.

- **명령어 수준의 병렬성(ILP : Instruction Level Parallelism)**
  - 프로그램을 구성하는 명령어들의 병렬처리 가능 정도를 의미.
  - 컴파일러와 하드웨어 기술을 적용하여 ILP 극대화 가능.

- **슈퍼스칼라 구조를 사용하는데 한계가 있다**
  - 데이터 의존관계, 프로그램 실행 흐름, 컴퓨팅 자원에 대한 동시 접근 문제 등은 ILP 를 높이는데 제한 요소로 작용.

2. Superscalar 제한 요소

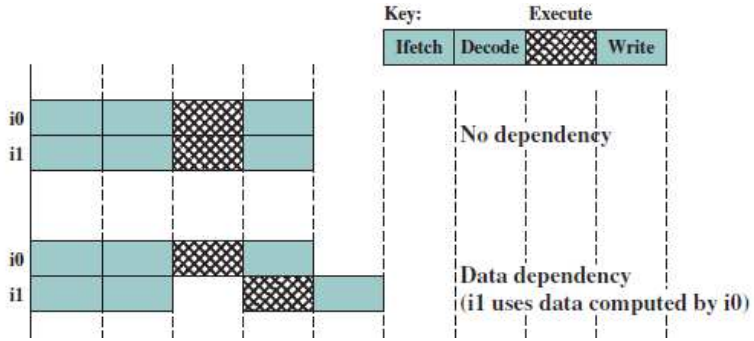
- **ILP 한계 요소 유형**
  - 데이터 의존성(Data Dependency)
    - 데이터 의존관계에 따라, 명령어의 동시 실행이 제한된다.
      - True data dependency
      - Anti-dependency
      - Output dependency
  - 실행흐름 의존성(Procedural Dependency)
    - 명령어 실행흐름에 따른 의존관계에 따른 제한 요소
  - 자원 충돌(Resource Conflicts)
    - 컴퓨팅 자원 접근 충돌에 의한 제한 요소

2.1 True Data Dependency

- **True data dependency**
  - Read After Write(RAW) 의존성
    - 첫 번째 명령어 실행 결과가 생성될 때까지 다음 명령어 실행이 시작될 수 없다.

- **문제 발생 상황**
  - 첫 번째 명령어가 레지스터나 메모리에 저장된 데이터 값을 변경 (Write)한 후,
  - 다음 명령어가 해당 데이터 값을 참조(Read)할 경우,
  - 데이터 값을 변경이 완료되기 전에 참조할 경우에는, 문제 발생(Hazard)

```
ADD r1, r2  // r1 = r1 + r2;
MOV r3, r1 // r3 = r1;
```



2.2 Anti-Dependency

- **Anti-dependency**
  - Write After Read(WAR) 의존성

- **문제 발생 상황**
  - 첫 번째 명령어가 레지스터나 메모리에 저장된 데이터 값을 참조(Read)한 후,
  - 다음 명령어가 해당 데이터 값을 변경(Write)할 경우,
  - 데이터 값을 참조하기 전에 다음 명령어에 의해 값이 변경될 경우, 문제 발생(Hazard)

```
ADD r1, r2  // r1 = r1 + r2;
SUB  r2, r3  // r2 = r2 – r3;
```

2.3 Output Dependency

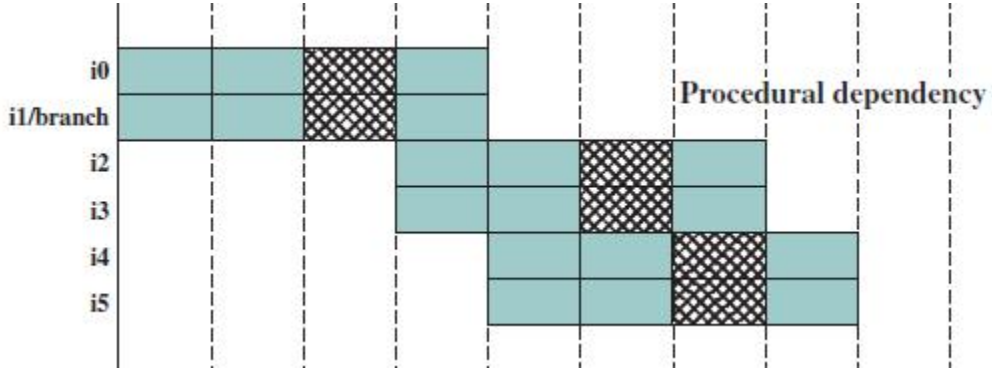
- **Output Dependency**
  - Write After Write(WAW) 의존성

- **문제 발생 상황**
  - 첫 번째 명령어가 레지스터나 메모리에 저장된 데이터 값을 변경(Write)한 후,
  - 다음 명령어가 해당 데이터 값을 다시 변경(Write)할 경우,
  - 첫 번째 명령어가 데이터 값을 변경하기 전에 다음 명령어에 의해 값이 변경될 경우, 문제 발생(Hazard)
    - 원래 실행 순서가 뒤바뀜

```
ADD r1, r2  // r1 = r1 + r2;
SUB  r1, r3  // r1 = r1 – r3;
```

2.4 실행흐름 의존성

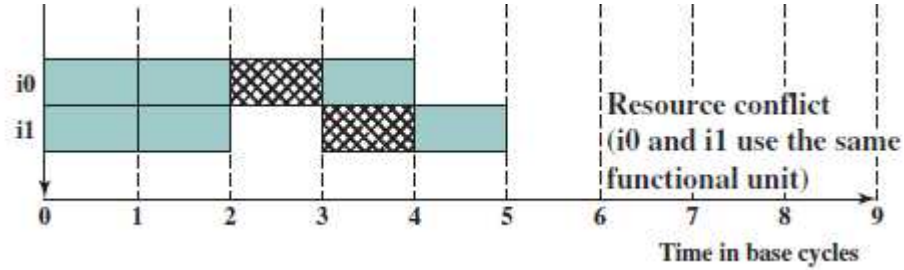
- **실행흐름 의존성(Procedural Dependency)**
  - 명령어 실행흐름 의존관계.
  - 분기 명령어 실행이 완료될 때까지 다음 명령어 실행을 시작할 수 없다.
  - 명령어 길이가 고정되어 있지 않은 경우, 명령어 fetch 횟수를 결정할 수 없어서 다른 명령어의 동시 fetch 를 방해하게 된다.



2.5 자원 충돌

- **자원 충돌(Resource Conflicts)**
  - 여러 개의 명령어들이 동일한 자원에 대한 동시 접근 시도할 경우, 문제발생.

- **자원 충돌 해결방법**
  - 컴퓨팅 자원을 여러 개 사용하여 충돌 방지
  - 실행시간이 긴 자원일 경우, 파이프라인을 적용하여 해결.



슈퍼스칼라 설계 요소

1. 설계 요소

- **명령어 실행속도 향상에 영향을 주는 요소**
  - 명령어 수준 병렬성(Instruction level parallelism)
    - 일련의 명령어들이 상호 독립적이어서 중첩실행이 가능한 경우 명령어 수준 병렬성이 높다.
    - 병렬성 정도는 데이터 및 실행흐름 의존관계 여부에 의해 결정된다.
  - 머신 수준 병렬성(Machine parallelism)
    - 명령어 수준 병렬처리 가능성을 활용할 수 있는 정도.
    - 병렬 실행 파이프라인 수에 의해 결정된다.

1. 설계 요소

- **명령어 실행속도 향상에 영향을 주는 요소 (계속)**
  - 명령어 발행 정책(Instruction issue policy)
    - 명령어 발행(Instruction issue)
      - 명령어 실행을 프로그램 실행회로에 지시하는 과정.
      - 명령어 해석이 끝난 후 실행 파이프라인의 첫 번째 실행단계로 진입하는 과정
    - 명령어 발생 정책(Instruction issue policy)
      - 명령어 발행 과정에 사용되는 프로토콜

2. 명령어 발행

- **명령어 발행(Issue) 전에 명령어 실행순서를 고려해야 한다.**
  - 명령어 실행 순서 (Instruction ordering)에 영향을 주는 요소
    - 명령어 인출(Fetch) 순서
    - 명령어 실행 순서
    - 데이터 값 (레지스터와 메모리에 저장된 값) 변경순서

- **프로세서는 어떠한 상황에서도 명령어 실행 결과는 정확해야 한다.**
  - 명령어 실행결과가 정확하게 보장된다면 실행속도 향상을 위해 실행순서를 변경할 수 있다.

3. 명령어 발행 정책

- **슈퍼스칼라에서 사용하는 명령어 발행 정책**
  - 명령어 발생시점과 완료시점에 따라 유형이 달라진다.
    - In-order Issue with In-order Completion
    - In-order Issue with Out-of-order Completion
    - Out-of-order Issue with Out-of-order Completion

3. 명령어 발행 정책

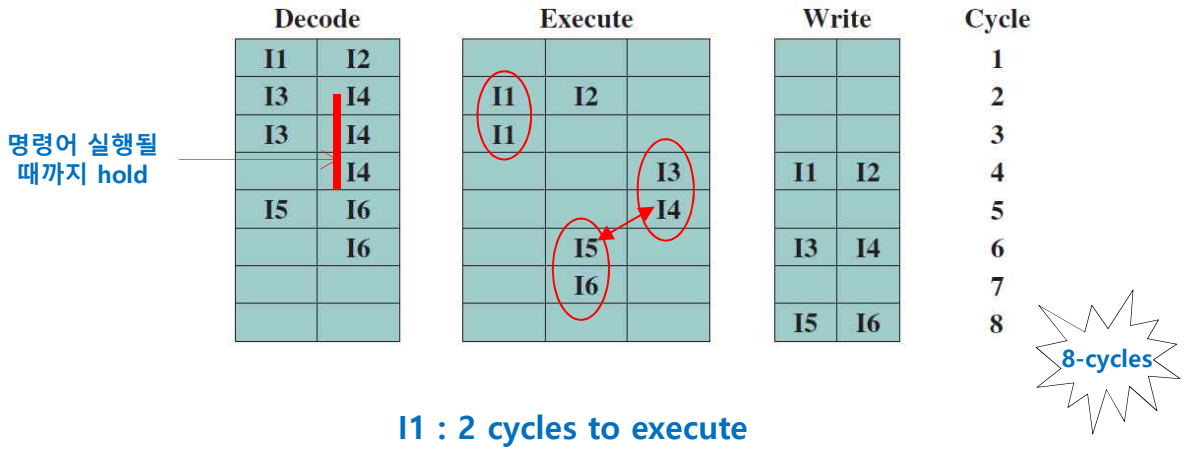
- **슈퍼스칼라에서 사용하는 명령어 발행 정책**
  - 명령어 발생 유형과 완료 유형에 따라 구분
    - In-order Issue with In-order Completion
    - In-order Issue with Out-of-order Completion
    - Out-of-order Issue with Out-of-order Completion

- **In-order Issue/Completion**
  - 프로그램에서 정의한 실행 순서대로 명령어 발행/실행 완료

- **Out-of-order Issue/Completion**
  - 프로그램에서 정의한 실행 순서와 다른 순서로 명령어 발행/실행 완료
  - 프로그램의 명령어 수준 병렬성(ILP) 정도에 따라, 명령어의 동시실행 가능
    - 데이터 의존성, 실행흐름 의존성, 자원충돌 문제를 해결해야 함.

3.1 In-order Issue with In-order Completion

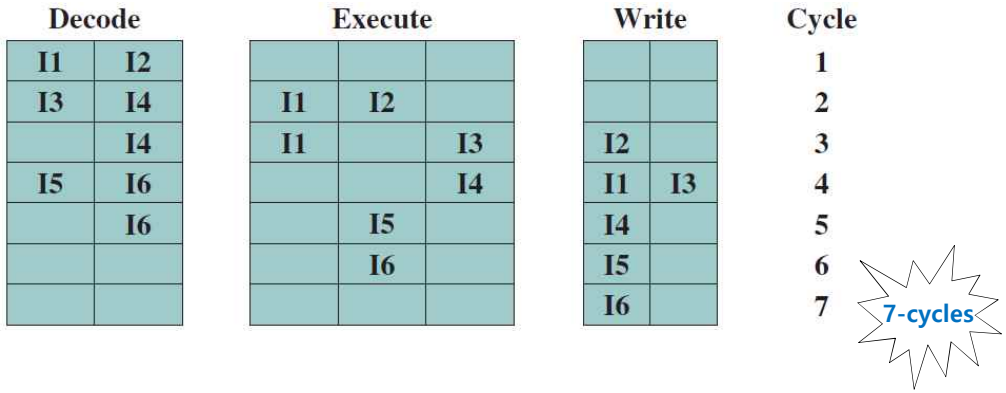
- **In-order issue with in-order completion 실행 예시**
  - Supercalar pipeline 구성
    - 2 fetch & decode pipelines
    - 3 functional units (2 integers, 1 floating)
    - 2 write-back pipelines



I1 : 2 cycles to execute
I3, I4 : conflict for same FU
I5, I6 : conflict for FU
I4, I5 : data dependency

3.2 In-order Issue with Out-of-order Completin

- **In-order issue with out-of-order completion 실행 예시**



I1 : 2 cycles to execute
I3, I4 : conflict for same FU
I5, I6 : conflict for FU
I4, I5 : data dependency

3.3 Instruction Issue Policy

- **Out-of-order issue with out-of-order completion 실행 예시**

Decode		Window	Execute			Write		Cycle
I1	I2							1
I3	I4	*I1, I2*	I1	I2				2
I5	I6	*I3, I4*	I1		I3	I2		3
		I4, I5, I6		I6	I4	I1	I3	4
		I5		I5		I4	I6	5
						I5		6

6-cycles

I1 : 2 cycles to execute
I3, I4 : conflict for same FU
I5, I6 : conflict for FU
I4, I5 : data dependency

4. 명령어 윈도우

- **명령어 윈도우(Instruction Window)**
  - 명령어 해석(Decode) 결과를 저장하는 버퍼

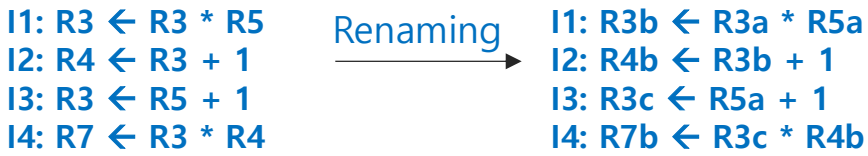
- **Out-of-Issue를 하려면, 명령어 윈도우를 사용하여 명령어 해석(Decode) 파이프라인과 실행(Execute) 파이프 라인이 완전 분리할 필요가 있음.**

- **명령어 윈도우 동작 절차**
  - 프로세서가 명령어 해석을 끝내면, 명령어 윈도우에 저장한다.
  - 명령어 윈도우가 비어 있으면, 프로세서는 다음 명령어를 fetch-decode 해서 그 결과를 계속 버퍼에 저장한다.
  - 실행 파이프라인이 비게 되면 명령어 윈도우에서 명령어를 꺼내 실행 파이프에 전달한다.

5. 레지스터 Renaming

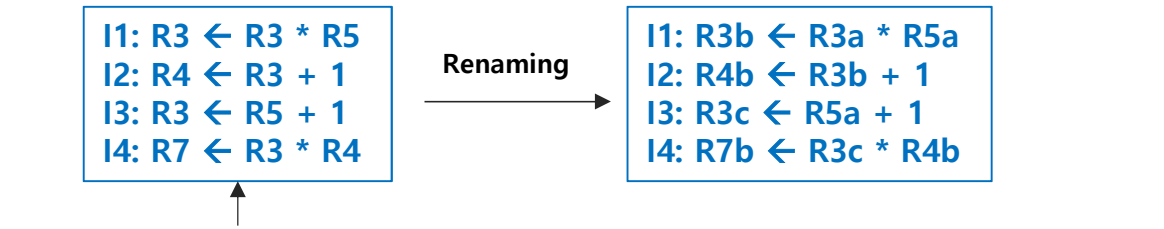
- **명령어 실행순서와 실행 결과, 데이터 변경(Write) 순서가 바뀌면 Output dependency 와 Anti-dependency를 검사할 필요가 있다.**
  - Output Dependency와 Anti-dependency 때문에 여러 개의 명령어들이 동일한 레지스터를 액세스하려는 저장 장소 충돌(Storage Conflict) 가 발생
  - 파이프라인이 멈추는 것(stall)을 방지하기 위해 레지스터를 하드웨어를 사용하여 동적으로 할당. → register renaming

- **Renaming**
  - 하드웨어적으로 레지스터 할당
  - 프로그램에서 정의한 레지스터가 아닌, 레지스터에 새로운 값이 생성(Write) 될 때 마다, 새로운 레지스터를 할당.



5. 레지스터 Renaming

- **Renaming 예시**



(Renaming 미사용)

* I3와 I1, I2사이의 의존성 때문에 I3 명령어는 I1이 종료되고, I2가 발행(Issue)될 때까지, I3 명령어 발행을 대기해야 함.

(Renaming 사용)
* I3 명령어에서 R3c 할당
- I2와의 Anti-dependency 방지
- I1과의 Output Dependency 방지
- I4는 R3c 사용

→ I3 명령어는 I1과 같이 발행(Issue) 가능

병렬처리 개요

1. 병렬처리 개요

- **단일 프로세서 한계**
  - 물리적 제약
  - 반도체 집적회로 공정상의 한계
  - 클럭속도 증가에 따른 소비전력 증가와 발열문제
  - 집적도 향상에 따른 RC Delay 증가

  - 해결방법
    - 다수의 프로세서를 사용하는 병렬처리 기법 적용한 성능향상

1. 병렬처리 개요

- **병렬처리**
  - 다수의 프로그램, 프로그램 조각, 명령어들을 다수의 프로세서에 분산시켜 동시에 실행함으로써 처리속도를 향상시키는 기술.

- **n 개의 프로세서를 사용함으로써 얻을 수 있는 이득**
  - 최대 n 배의 처리속도 향상
  - 소비전력 감소

2. 병렬처리 특성

- **프로그램 분할**
  - 프로그램을 큰 단위로 분할(Coarse-Grain)
    - 병렬성 저하
    - 동기화 및 스케쥴링 복잡도 감소
  - 프로그램을 작은 단위로 분할(Fine-Grain)
    - 병렬성 향상
    - 동기화 및 스케쥴링 복잡도 증가

2. 병렬처리 특성

- **프로세서 스케쥴링**
  - 분할된 프로그램을 프로세서에 효율적으로 배정해야 한다.

- **공유자원에 대한 접근 중재**
  - 프로세서 사이의 공유자원에 대한 접근충돌 중재

- **공유 데이터의 동기화**
  - 공유 데이터에 대한 일관성 유지 문제

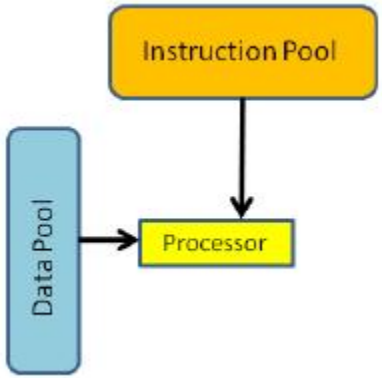
3. 병렬처리 시스템 분류

- **Flynn 의 분류법**
  - 명령어 스트림과 데이터 스트림을 기준으로 컴퓨터 시스템을 분류
    - 명령어 스트림 : CPU에 의해 수행되는 연속적인 명령어 흐름
    - 데이터 스트림 : 명령어 스트림 수행에 필요한 데이터의 연속적 흐름

- **분류 유형**
  - SISD (Single Instruction, Single Data)
  - SIMD (Single Instruction, Multiple Data)
  - MISD (Multiple Instruction, Single Data)
  - MIMD (Multiple Instruction, Multiple Data)

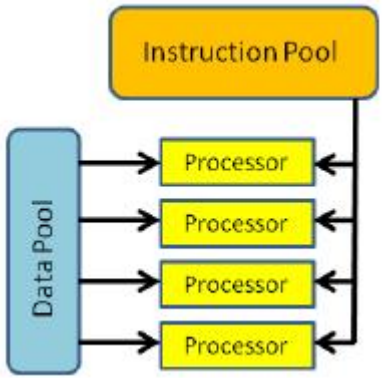
3.1 SISD

- **SISD**
  - 하나의 명령어 스트림으로 하나의 데이터 스트림을 한번에 처리하는 시스템
  - 단일 프로세서 구조 : 폰-노이만 구조
  - 파이프라인, 슈퍼스칼라를 사용해서 성능향상



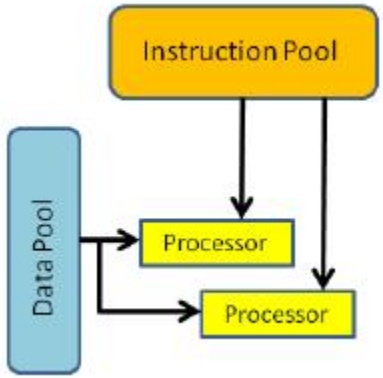
3.2 SIMD

- **SIMD**
  - 하나의 명령어 스트림으로 다수의 데이터 스트림을 동시에 처리하는 시스템
  - 데이터 수준의 병렬성을 이용
  - 하나의 제어장치 + 다수의 처리장치(Processing Element)로 구성
  - 벡터 프로세서, Array Processor
  - 멀티 미디어 데이터 처리에 적합



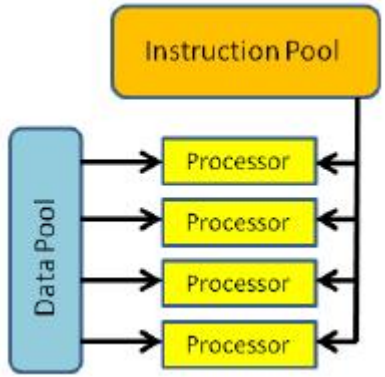
3.3 MISD

- **MISD**
  - 다수의 명령어 스트림이 하나의 데이터를 처리하는 시스템
  - 실용성 부족 : 구현사례 없음

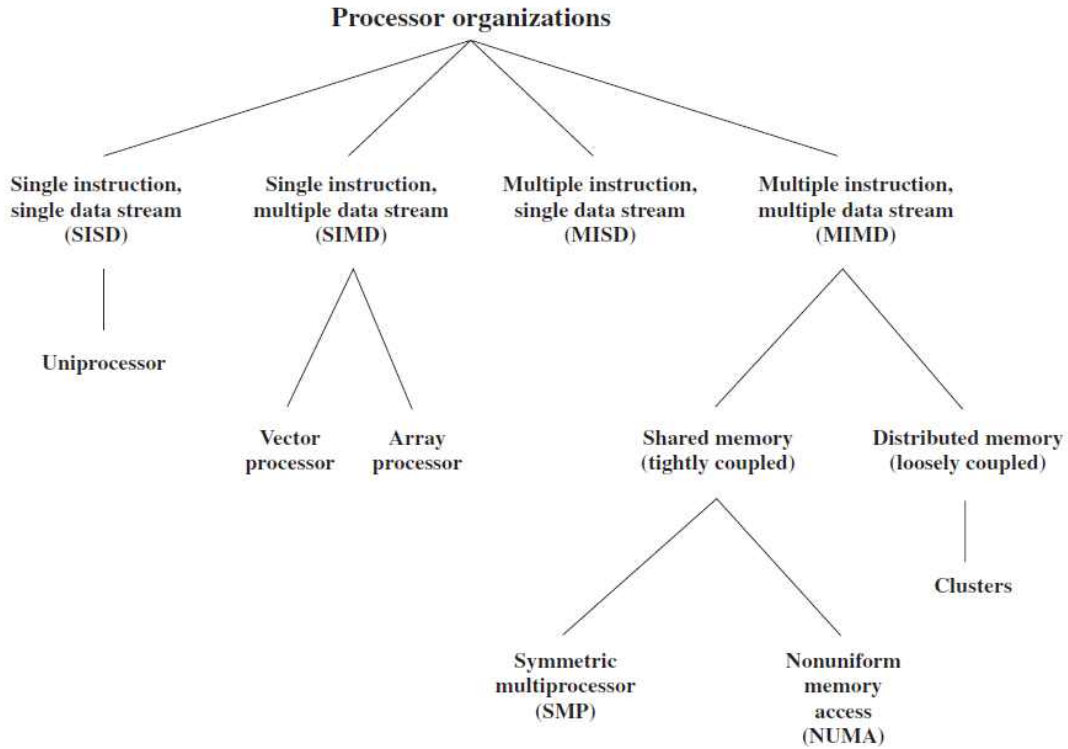


3.4 MIMD

- **MIMD**
  - 다수의 명령어 스트림이 다수의 데이터를 동시에 처리하는 시스템
  - 대부분의 병렬처리 시스템
  - 다중 프로세서 시스템(Multiprocessor System)
    - 여러 개의 CPU를 사용한 하나의 컴퓨터
  - 다중 컴퓨터 시스템(Multi-Computer System)
    - 여러 개의 독립적인 컴퓨터를 사용하여 구성한 하나의 시스템

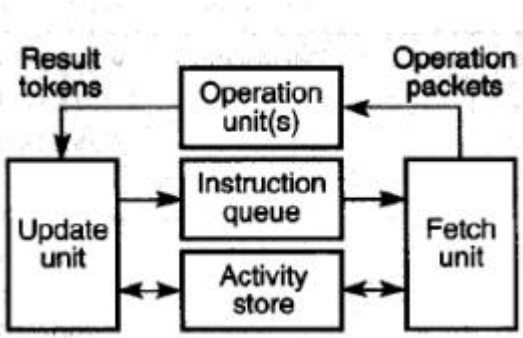


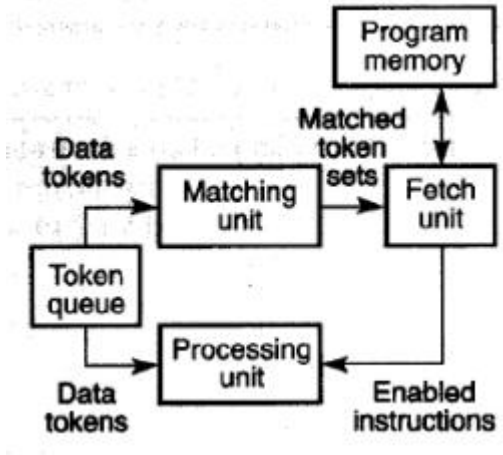
3.5 병렬 프로세서 분류


Processor organizations
Single instruction, single data stream (SISD)
Single instruction, multiple data stream (SIMD)
Multiple instruction, single data stream (MISD)
Multiple instruction, multiple data stream (MIMD)
Uniprocessor
Vector processor
Array processor
Shared memory (tightly coupled)
Distributed memory (loosely coupled)
Clusters
Symmetric multiprocessor (SMP)
Nonuniform memory access (NUMA)

3.6 Dataflow Computer

- **데이터와 직접 연결하여 프로그램 실행을 제어**
  - Control Flow 구조인 기존의 폰-노이만 구조와 반대
  - 프로그램 카운터가 없음
  - 실행에 필요한 데이터가 준비된 경우에만 명령어 실행
  - 명령어 실행순서는 데이터 의존성에 의해 결정.

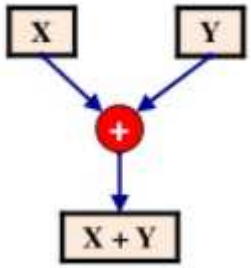




SIMD/MIMD 프로세서

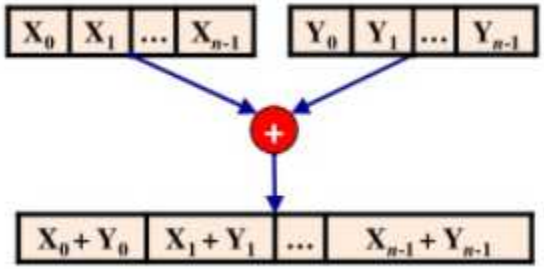
1. SIMD 프로세서

- **벡터 프로세서(Vector Processor)**
  - 한번에 여러 개의 데이터(벡터)를 처리할 수 있는 프로세서
    - 한번에 하나의 데이터를 처리하는 스칼라(Scalar) 프로세서와 대비.
    - Array 프로세서



ADD Z, X, Y

Scalar Processor



ADD Z[n-1:0], X[n-1:0], Y[n-1:0]

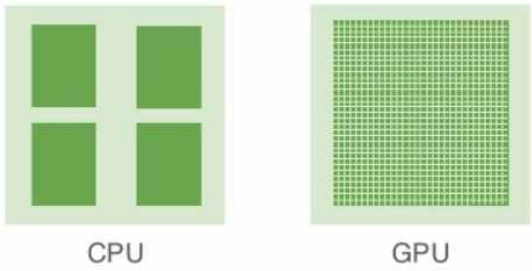
Vector Processor

1. SIMD 프로세서

- **벡터 프로세서 구성**
  - 하나의 제어장치 + 여러 개의 처리장치(CPU+레지스터) + 전용 메모리

- **컴파일러 역할 중요**
  - 컴파일러가 프로그램을 분석해서 SIMD 처리가 가능한 부분은 SIMD 명령어로 대체하는 작업(Vectorization)을 수행.

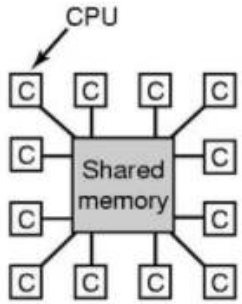
- **대표적 벡터 프로세서**
  - GPGPU : 그래픽 처리를 위해 벡터 프로세서를 사용.

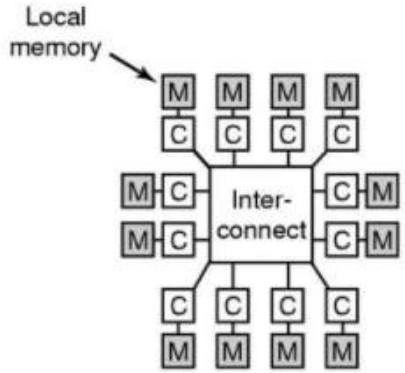


2. 다중 프로세서 시스템

- **다중 프로세서 시스템**
  - 다수의 프로세서로 구성된 하나의 컴퓨터
  - 다수의 프로세서가 컴퓨터 자원을 공유 → Shared Memory Multi-processor
  - 하나의 작업을 수행하는데 다수의 프로세서가 협조(공조) → Tightly Coupled System

- **다중 프로세서 시스템 유형**
  - UMA(Uniform Memory Access)
  - NUMA(Non-Uniform Memory Access)
  - 멀티코어(Multi-Core)





2. 다중 프로세서 시스템

- **다중 프로세서 시스템 장점**
  - 프로세서 사이의 데이터 교환 메커니즘이 필요 없다.
  - 프로세서에 작업을 균등하게 배분할 수 있어서 프로세서 이용률 향상
  - 모든 프로세서가 단일 주소 공간을 사용하기 때문에 기존 프로그램에 쉽게 적용가능.

- **다중 프로세서 시스템 단점**
  - 프로세서와 메모리 사이의 트래픽 증가 : 높은 대역폭 버스 필요
  - 다수의 프로세서가 동일 메모리 공간에 동시 접근시 충돌 발생
    - 각 프로세서가 전용 캐시 사용해서 해결

2. 다중 프로세서 시스템

- **다중 프로세서 시스템에서의 캐시 사용 문제점**
  - 데이터에 대한 일관성(Coherence) 문제 발생
  - 하드웨어적 문제 해결 방법 : Snoopy 또는 Directory 프로토콜 사용

- **스누피(Snoopy) 프로토콜**
  - 모든 캐시 컨트롤러가 다른 캐시 컨트롤러를 모니터링.
  - 공유블럭이 수정될 때 전체 캐시에 알려주는 Broadcasting 방식 프로토콜

- **디렉토리(Directory) 프로토콜**
  - 캐시의 상태정보를 저장하고 있는 디렉토리를 중앙 제어기가 관리.
  - 캐시 컨트롤러는 다른 캐시 컨트롤러를 모니터링하지 않고, 디렉토리와 정보교환.
  - 공유블럭이 갱신될 때마다 디렉토리에 표시하는 방식

2.1 UMA

- **다중 프로세서 시스템은 메모리 접근시간 유형에 따라 UMA 구조와 NUMA 구조로 구분**

- **UMA(Uniform Memory Access) 구조**
  - 중앙 집중식 공유 메모리 사용
  - 모든 프로세서들은 모든 메모리 공간에 동일한 접근 시간으로 접근할 수 있음.
  - 구조가 간단하고 프로그램이 쉽다.
  - 프로세서 수가 증가하면 메모리 접근 트래픽이 과도하게 증가

- **UMA 유형**
  - SMP(Symmetric Multi-Processor)
    - 같은 종류의 프로세서 사용
    - 메모리 풀(Pool)과 주변장치를 완전히 공유하는 구조

2.2 NUMA

- **NUMA(Non-Uniform Memory Access) 구조**
  - 하나의 연속적인 공유 메모리 공간을 모든 프로세서에 분산시키는 구조
    - 분산 공유 메모리 구조
  - 단점
    - 프로그램 코딩이 어려움
    - 캐시 일관성 확보 어려움

- **CC-NUMA (Cache Coherent NUMA)**
  - 캐시 일관성 문제를 해결한 NUMA

2.3 멀티코어

- **다수의 CPU 코어를 물리적으로 하나의 집적회로(IC)에 집적시킨 다중 프로세서 시스템**
  - 동일 칩 (chip) 상에 여러 개의 CPU 코어들을 집적한 프로세서
  - 여러 개의 CPU 코어가 큰 용량의 캐시를 공유.
  - 클럭 속도를 높이지 않고도 성능을 향상 가능.

- **프로그램이 여러 개의 프로세서에서 효과적으로 실행된다면, 프로세서 수를 2배 늘리면, 성능도 거의 2배로 향상된다.**

- **프로세서 내부의 성능 향상은 복잡도 증가의 제곱근에 비례한다고 알려짐.**

2.3 멀티코어

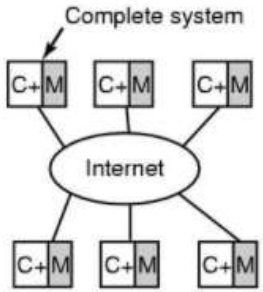
- **동종 멀티 코어(Homogeneous Multi-Core) 프로세서**
  - 동일한 코어를 사용한 프로세서
  - CPU + CPU + CPU

- **이종 멀티 코어(Heterogeneous Multi-Core) 프로세서**
  - 서로 다른 코어를 사용한 프로세서
  - CPU + GPU + DSP

3. 다중 컴퓨터 시스템

- **프로세서 수준에서 구현된 다중 프로세서 시스템과 달리 시스템 수준에서 구현된 시스템**
  - 연결된 컴퓨터의 결합강도가 약하기 때문에 Loosely Coupled System 이라고 함.
  - 메모리 공유가 없이 개별 메모리를 사용 → 분산 메모리 컴퓨터 시스템
  - 메시지 전달(Message Passing)을 통한 데이터 공유
    - 공유 데이터가 증가하면 노드사이의 통신량 증가 → 성능저하
  - 지리적으로 분산된 여러 개의 독립된 컴퓨터(Node)로 구성

- **다중 컴퓨터 시스템 유형**
  - 클러스터(Cluster)



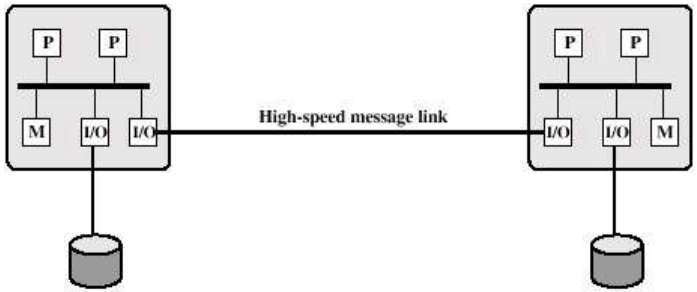
3. 다중 컴퓨터 시스템

- **장점**
  - 각 프로세서마다 전용 메모리를 사용하기 때문에 버스 경합이 없다.
  - 공유 버스를 사용하지 않기 때문에 노드 확장성이 좋다.
  - 캐시 일관성 유지 문제가 없다.

- **단점**
  - 빈번한 메시지 전송에 따른 성능저하.
  - 독립적 프로그램으로 구성해야 하기 때문에 프로그래밍 부담이 증가한다.

4. 클러스터

- 독립된 컴퓨터들을 네트워크로 연결하여 구성
- 클러스터를 구성하는 각 컴퓨터는 서로 다른 운영체제를 사용할 수 있다.
- 저가의 컴퓨터 시스템을 네트워크로 연결하여 하나의 컴퓨터로 사용
- 많은 양의 데이터를 처리할 때 유리.
- 작업을 작은 규모의 소단위 작업으로 분할할 수 있는 작업에 적합.



4. 클러스터

- **장점**
  - 낮은 구현비용 : 저렴한 컴퓨터를 네트워크로 상호연결. 상대적으로 저렴한 구현 비용
  - 높은 가용성(Availability) : 노드 고장시 대체노드 사용이 용이. 장애극복능력 우수
  - 확장성 : 네트워크 연결을 통한 확장이 용이
  - 유연성 : 다양한 환경 구축이 가능
  - 고성능 : 다수의 컴퓨터에 작업을 분할처리 응답속도 향상

- **단점**
  - 높은 관리 비용 : 관리 노드 수 의 증가
  - 네트워크 지연 : 네트워크 지연시간에 의한 성능저하.

4. 클러스터

- **클러스터 통신 방식**
  - MPI(Message Passing Interface) 프로토콜 사용.
  - PVM(Parallel Virtual Machine) 형태로 상호통신.

- **PVM**
  - 컴퓨터의 병렬 연결을 위한 소프트웨어
  - 서로 다른 운영체제를 사용하는 컴퓨터들을 연결하여 하나의 분산형 병렬 컴퓨터로 사용할 수 있게 해준다.

- **MPI**
  - 소켓을 사용한 표준 메시지 전달 방법
  - Pint-to-Point 방식의 메시지 전달

상호 연결망 구조

1. 상호 연결망 구조

- **상호 연결망 구조(Interconnection Network)**
  - 다중 프로세서, 다중 컴퓨터 시스템은 상호 연결망 을 통해서 연결된다.

- **연결망 분류기준**
  - 동기화 방식
    - 전역 클럭 사용여부에 따라 동기식, 비동기식
  - 데이터 교환방식
    - 회선 교환(Circuit Switching) : 노드 사이의 물리적 통신경로를 설정한 후 데이터 전송. 긴 메시지의 간헐적 교환에 적합.
    - 패킷 교환(Packet Switching) : 물리적 경로 설정없이 메시지를 패킷 단위로 분할 전송. 짧은 메시지의 반복적 교환에 적합.

1. 상호 연결망 구조

- **연결망 분류 기준(계속)**
  - 제어방식
    - 중앙 집중식 : 중앙 제어기가 메시지 교환 요청을 처리
    - 분산 제어식 : 연결망에 연결된 프로세서 또는 노드 컴퓨터가 자체적으로 메시지 전송 처리
  - 연결 형태
    - 토폴로지(Topology) : 노드 사이의 연결방식
    - 정적 토폴로지 : 노드 연결이 고정, 재구성 불가, 전용링크 사용
    - 동적 토폴로지 : 노드 연결이 유동적, 재구성 가능, 네트워크 스위치 사용

2. 정적 상호 연결망

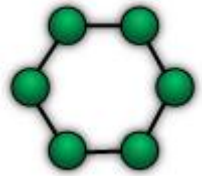
- **노드 사이가 직접 연결되고 연결이 고정된 형태**
- **연결 형태 구분**
  - 노드의 Degree와 연결망의 Diameter에 따라 구분.
  - 노드의 Degree
    - 노드에 연결된 링크 수
  - 연결망의 Diameter
    - 송수신 노드 사이의 최단 연결에 소요되는 링크 수
    - 연결망을 통해 메시지를 전달하는데 걸리는 지연시간을 결정

2. 정적 상호 연결망

- **1차원 토폴로지**
  - 선형 배열구조
    - N개의 노드가 순차적으로 1차원적으로 배열된 연결망.
  - 평균 통신시간이 가장 길다.
  - 구조가 단순
  - 버스구조에 비해 동시성이 높다.
  - 노드 수가 증가하면 평균 통신시간이 길어진다.

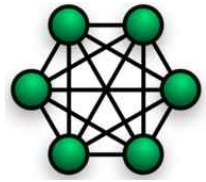
2. 정적 상호 연결망

- **2차원 토폴로지**
  - 노드들을 2차원적으로 연결한 구조
  - Ring, Tree, Mesh, Star 구조



- **Ring 구조**
  - 지연시간이 길고 확장성이 떨어진다.

- **Tree 구조**
  - 루트(Root) 노드에 가까운 상위 노드일수록, 트래픽이 집중된다.

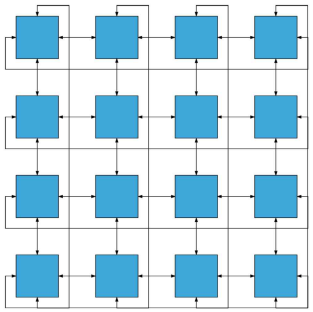


- **Mesh 구조**
  - 각 노드를 2차원적으로 배열한 구조

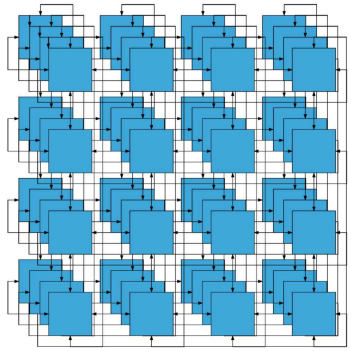
- **Star 구조**
  - 중앙노드(Hub Node)와 Point-to-Point 로 연결된 구조.

2. 정적 상호 연결망

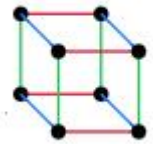
- **다차원 토폴로지**
  - 노드들을 다차원적으로 연결한 구조
  - Torus 구조 : Mesh + Ring 구조
  - Cube 구조 : 노드를 3차원으로 배치한 구조



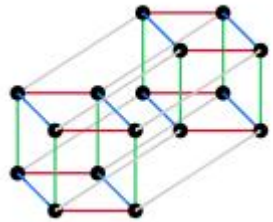
2D Torus



3D Torus



3D Cube



4D Cube

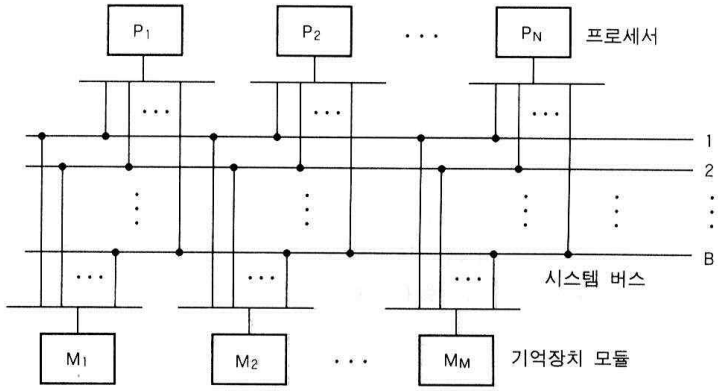
2. 동적 상호 연결망

- **노드 사이의 데이터 경로가 시간에 따라 다양하게 변경될 수 있는 연결망**
  - 노드사이의 데이터 경로에 대한 경합 발생

- **상호 연결방식에 따른 분류**
  - 버스 기반 연결망 : 간단하고 저비용, 버스 공유에 따른 충돌발생
  - 스위치 기반 연결망 : 높은 연결성, 고비용

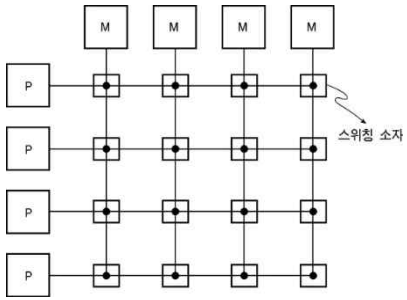
2. 동적 상호 연결망

- **버스기반 연결망**
  - 다중 버스 구조
    - 여러 개의 버스를 사용하여 독립적인 통신 채널 확보 구조



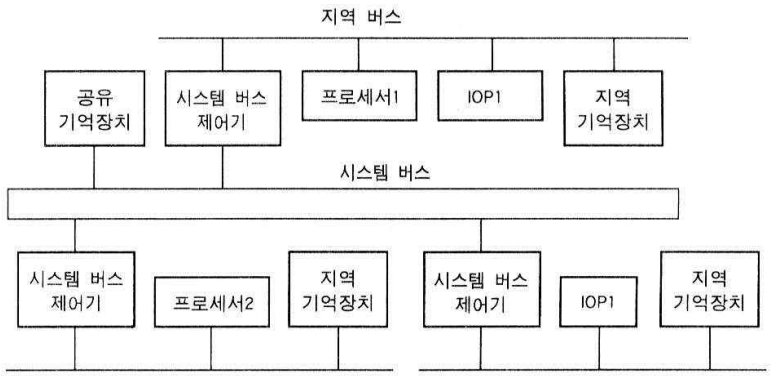
2. 동적 상호 연결망

- **버스기반 연결망**
  - 크로스바(Crossbar) 스위치
    - 교차점에 스위칭 소자를 사용하여 연결
    - 통신 경로의 경합 최소화
    - 네트워크 비용 최대



2. 동적 상호 연결망

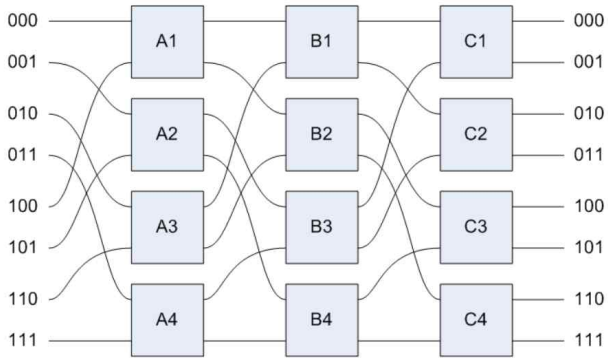
- **버스기반 연결망**
  - 계층 버스 구조
    - 버스 특성(통신속도)에 따라 버스를 계층구조로 연결



2. 동적 상호 연결망

- **스위치 기반 연결망**
  - 입력과 출력사이에 스위칭 소자를 사용한 연결 망

- **오메가(Omega) 네트워크**
  - 다단계(Multi-Stage) 스위치를 사용한 연결망 구성.
  - 각 단계(Stage)에서의 연결은 셔플(Shuffle) 방식으로 연결



8x8 Omega Network

end