이것이 취업을 위한 컴퓨터 과학이다 CH.02 컴퓨터 구조

컴퓨터 부품

- CPU / 버스 / 메모리 / 입출력장치

CPU

- ALU
  - 산술논리연상장치
  - Arithmetic Logic Unit
  - 일종의 계산기
- 제어장치
  - CU
  - Control Unit
  - 제어신호를 내보내고, 명령어를 해석
    - 컴퓨터 부품에게 명령을 내리기 위한 전기신호
- 레지스터
  - CPU 내부의 임시 저장장치
  - 레지스터 8가지 각각의 역할
    - 프로그램 카운터
      - 메모리에서 읽어 들일, 다음으로 실행할 "명령어의 주소"
        - instruction Pointer
        - 명령어 포인터, 라고 부르는 CPU도 있음
      - 메모리 버퍼 레지스터 까지 사용했으면 카운터가 1 늘어남
        - 이게 프로그램이 순차적으로 실행될 수 있는 원리이고 이유임
      - 순차적인 실행흐름이 끊기기도 함

        - 특정 메모리 주소로 실행 흐름을 이동하는 명령어 실행 시
            - JUMP, CONDITIONAL JUMP, CALL, RET
        - 인터럽트 발생 시
        - ETC...
    - 상대 주소 지정방식
        - 오퍼랜드 필드의 값(변위)과 "프로그램 카운터"의 값을 더하여 유효주소 얻기
        - 이 다음번 실행할 명령어의 주소(=프로그램 카운터)에 값을 연산하면 내가 원하는 유효주소를 얻을 수 있다
- 메모리 주소 레지스터
    - 메모리의 주소를 저장
    - CPU가 내보낼/들여보낼 주소를
    - 주소 버스로 보낼 때
    - 거치는 레지스터
- 메모리 버퍼 레지스터
    - 메모리와 주고받을 값 = 데이터와 명령어
    - CPU가 정보를
    - 데이터 버스로 주고받을 때
    - 거치는 레지스터
- 명령어 레지스터
    - 제어장치가 해석할 명령어 = 방금 메모리에서 읽어 들인 명령어
    - 이거+ALU+제어장치 = 연산
- 이상은 순서대로 사용됨
- 플래그 레지스터
    - 연산결과 or CPU 상태에 대한 부가적인 정보
- 범용 레지스터
    - 다양하고 일반적인 상황에서 자유롭게 사용
- 스택 포인터

        - 특별한 주소지정에 사용됨-1
        - 스택의 꼭대기를 가리키는 레지스터
        - 스택이 어디까지 차있는지에 대한 표시
            - 스택은 메모리 안에 있음
            - 스택포인터가 4면 스택이 4까지 차있다는 뜻
        - 스택 주소 지정방식
            - 스택과 스택 포인터를 이용한 주소 지정방식
    - 베이스 레지스터
        - 특별한 주소지정에 사용됨-2
        - 기준주소 저장
        - 변위 주소 지정방식
            - 오퍼랜드 필드의 값(변위)과 "특정 레지스터"의 값을 더하여 유효주소 얻기
            - 특정 레지스터 = 베이스 레지스터
            - 베이스 레지스터에는 기준주소가 담여있음
                - 이것을 대상으로 변위를 연산하면 유효주소를 얻게 되는 것

버스

- 컴퓨터의 부품끼리 정보를 주고받는 일종의 통로
    - 시스템 버스
        - 주소버스 / 데이터 버스 / 컨트롤 버스
    - 로컬 버스
        - 시스템 버스의 클럭 속도보다 저속으로 동작하는 장치를 제어하는 신호선
    - 둘은 브릿지로 연결됨

메모리

- 주기억장치
    - 우선, 마이크로 컨트롤러 = CPU + 메모리 + I/O 장치
    - ROM

        - 마이크로 컨트롤러에 "부여하는 프로그램"을 저장한 메모리
        - 바이오스 등
    - RAM
        - 마이크로 컨트롤러가 프로그램을 "실행하기 위해 일시적"으로 사용하는 메모리
        - 휘발성
    - 캐시 메모리
        - 더 비싸고 더 가깝고 더 빠름
- 보조기억장치
    - 비 휘발성

입출력장치

- 마우스 키보드 모니터 등

---

etc

MB vs MiB

- 1000kB vs 1024 kiB

워드 work

- CPU가 한 번에 처리할 수 있는 정보의 크기 단위
- 64비트 컴퓨터 vs 32비트 컴퓨터 vs 8비트 컴퓨터

숫자표현

- **16진법**
- **부동소수점**
    - IEEE754
        - 부호 1비트 + 지수 8비트 + 가수(소수부분) 23비트

문자표현

- **문자집합 (Character Set)**
  - **고정 길이 인코딩**
    - 아스키
      - 코드 포인트 = `A -> 65`
    - EUC-KR
  - **가변 길이 인코딩**
    - UTF-8
    - UTF-16
    - UTF-32
- **01 -> 문자**
  - 인코딩
- **문자 -> 01**
  - 디코딩

명령어

- **무엇을 대상으로 무엇을 수행하라**

연산코드

- **수행할 연산**
  - 연산코드의 종류와 생김새는 CPU마다 다름
- **기본적으로 4가지**
  - 어셈블리 명령어 자체를 외울 필요는 없음
    - CPU마다 상이하기 때문
    - 역할은 알아야 함
  - **데이터 전송**
  - **산술/논리 연산**
  - **제어 흐름 변경**
  - **입출력 제어**

오퍼랜드

- **주소필드**
- **연산에 사용될 데이터**
  - **or**
- **연산에 사용될 데이터가 저장된 주소**
- 명령어 안에서 0~N개일 수 있음
- 데이터보다 주소가 훨씬 자주 담김
  - 그래서 "주소필드"라고 부르기도 함
- **주소를 더 자주 담는 이유**
  - 명령어의 총 데이터 크기가 제한되어 있기 때문
  - 오퍼랜드가 많아지면 오퍼랜드가 담을 수 있는 크기를 나눠 써야 함
    - 주소를 담으면 결국 더 많은 데이터를 다룰 수 있음

명령어 사이클

- 프로그램 속 명령어들은 일정한 주기가 반복되며 실행
- CPU는 명령을 일정한 흐름대로 처리함
- 그 주기 = **명령어 사이클**

명령어 사이클

- 아래 2~4가지를 반복:
  1. **인출 사이클**
     - 가장 먼저 CPU로 가져와야 함
  2. **실행 사이클**
     - 가져온 명령어를 실행해야 함
  3. **간접 사이클**
     - 메모리 접근이 더 필요한 경우 발생
     - 일부 명령어는 "인출+실행" 사이클만으로 실행됨
     - 일부 명령어는 "인출+간접+실행" 사이클을 거침

- 일부 명령어는 "인출+간접+실행+인터럽트" 사이클을 거침

인터럽트 사이클

- **인터럽트**: 명령어 사이클을 끊는 명령어

동기 인터럽트

- 예외사항 (e.g., CTRL+C)
- CPU가 예기치 못한 상황을 접했을 때 발생
  - "어라, 이 예외적인 상황은 뭐지?"
  - "잠깐 실행을 중단하고 이걸 처리해야겠다!"
- **종류**
  - 폴트
  - 트랩
  - 중단
  - 소프트웨어 인터럽트

하드웨어 인터럽트

- **비동기 인터럽트**
  - 주로 입출력장치에 의해 발생
  - **역할**:
    - "이런 입력이 들어왔으니" - 입력장치
    - "처리해 주세요" - CPU
  - 입출력 작업 도중 효율적 처리를 위해 사용
  - 인터럽트 없을 경우:
    - CPU는 프린트 완료 여부를 주기적으로 확인해야 함
    - 입출력장치는 CPU보다 느림
  - 인터럽트 있을 경우:
    - 입출력 작업 중 CPU는 다른 작업 가능
    - "난 다른 일 하고 있을 테니, 입출력이 완료되면 불러줘!"

- **인터럽트 종류**
  - 막을 수 있는 인터럽트: **maskable interrupt**
  - 막을 수 없는 인터럽트: **non-maskable interrupt**

인터럽트 처리 순서

- 모든 인터럽트는 대체로 동일한 절차로 처리됨:
  1. 입출력장치가 CPU에 인터럽트 요청신호를 보냄
     - "지금 끼어들어도 되나요?"
  2. CPU는 실행 사이클이 끝나고 명령어를 인출하기 전 인터럽트 여부 확인
     - 플래그 레지스터의 **인터럽트 플래그**를 확인
     - 모든 인터럽트를 플래그로 막을 수는 없음
  3. CPU는 인터럽트 요청을 확인하고, 플래그로 인터럽트 수락 여부 판단
     - **maskable / non-maskable**
  4. CPU는 인터럽트를 수락하면 현재 작업을 백업
     - 스택 영역에 백업
     - 프로그램 카운터도 백업/복구 과정에서 이동
  5. CPU는 **인터럽트 벡터**를 참조하여 인터럽트 서비스 루틴 실행
     - **인터럽트 벡터**:
       - 각각의 인터럽트를 구분하는 정보
       - "이 루틴의 시작점은 여기구나"를 알려줌
       - 인터럽트 요청신호와 함께 데이터 버스로 전달됨
     - **인터럽트 서비스 루틴**:
       - 인터럽트 발생 시 처리할 프로그램
       - 각 인터럽트는 서비스 루틴의 시작 주소를 메모리에 저장
  6. 서비스 루틴 실행 후, 백업된 작업 복구 후 실행 재개

인터럽트 플래그

- **플래그 레지스터**에 저장

- 인터럽트가 가능한지 여부:
  - `1`: 인터럽트 가능
  - `0`: 인터럽트 불가능
- **non-maskable interrupt**:
  - 이 플래그로 막을 수 없는 인터럽트

예외

- == 동기 인터럽트
  - 폴트 / 트랩 / 중단 / 소프트웨어 인터럽트

폴트

- 예외를 처리한 다음, 예외가 발생한 명령어부터 실행
- 예시, 페이지 폴트

트랩

- 예외가 발생한 명령어의 다음 명령어부터 실행
- 예시, 디버깅의 브레이크 포인트

중단

- CPU가 프로세스를 강제로 중단시킬 수 밖에 없는 심각한 오류를 발견했을때 방생하는 예외

소프트웨어 인터럽트

- 시스템 콜의 상황에서 발생하는 예외

CPU 성능향상을 위한 설계

클럭

- 컴퓨터의 부품이 움직이는 시간단위

    - 컴퓨터 부픔들은 '클럭 신호에 맞추니 일시불란하게 움직인다.
    - CPU는 '명령어 사이클이라는 정해진 흐름에 맞춰 명령어들을 실행한다.
- 그럼 클럭 신호가 빠르게 반복되면 CPU를 비롯한 컴퓨터 부품들은 그만큼 빠르게 움직이는가?
    - 일반적으로 YES
- 클력 속도
    - 해르츠 단위로 측정
- 헤르츠Hz
    - 1초에 클력이 반복되는 횟수
- 1Hz
    - 클력이 1초에 1번 반복
- 100Hz
    - 클력이 1초에 100번 반복
- 그럼 클럭 신호를 마냥 높이면 CPU가 무조건 CPU가 빨라지나?
    - NO, 필요 이상으로 클럭을 높이면 발열이 심각해짐
        - 쓰로틀링을 시켜서 온도를 안낮추면 녹아서 고장남
    - 그래서 클럭속도를 높이는 방법 이외에 이하의 방법이 필요한것

멀티코어

- 현대적인 관점에서 "CPU"라는 용어를 재해석 해야 함
- '명령어를 실행하는 부품'?
    - 전통적으로 '명령어를 실행하는 부품'은 원칙적으로 하나만 존재
- But 오늘날 CPU에는 '명령어를 실행하는 부품'이 여러개 존재
    - ALU+레지스터+제어장치 세트가 여러개 있는게 오늘날의 CPU
- 코어 = 명령어를 실행하는 부품
- 얼티코어 프로세서 = 코어가 여러개인 CPU
- 코어를 계속 늘린다고 연산속도가 따라서 빨라지나?
    - NO, 꼭 코어 수에 비례하여 증가하지는 않음

        - 정비례는 아닌데 아무튼 드락마틱하게 증가는 함
    - 요리사 5명이 도시락 1개 준비하나 요리사 10명이 도시락 1개 준비하나 비례하는 속도차이가 발생하지는 않듯이
    - 분배가 중요하다는 얘기

멀티 스레드

- 스레드 = 실행흐름의 단위
    - 하드웨어적 스레드
        - 하나의 코어가 동시에 처리하는 명령어 단위
        - 논리 프로세서라고도 부른다
            - 눈에 안보이기 때문
            - 실제 프로세서의 갯수는 아니지만
                - 메모리에서 실행되고 있는 프로그램이 느끼기에
                - 몇개의 CPU가 있느냐 하는 것
        - 코어 1개가 2스레드를 실행하고 코어가 2개면,
            - 2코어 4스레드 CPU가 되며,
            - 멀티 스레드 프로세서,
            - 멀티스레드 CPU이다
        - 멀티스레드 프로세서를 실제로 설계하는 일은 매우 복잡하지만,
            - 가장 큰 핵심은 레지스터
        - 레지스터 세트
            - "1개의 명령어를 실행하기 위해 꼭 필요한 레지스터들"
            - 을 편의상 지금 이렇게 표기하겠음
        - "레지스터 세트"가 "코어"안에 n개 있으면
            - 명령어를 n개 처러할 수 있음
    - 소프트웨어적 스레드
        - 하나의 프로그램에서 독립적으로 실행되는 단위
        - 소프트웨어적 멀티 스레드

        - 하나의 프로그램에서 두개 이상의 영역이 동시에 실행된다
    - 1코어 1스레드 CPU도 소프트웨어적 멀티 스레드를 만들 수 있다

명령어 병렬처리 기법

- 설계도 중요하지만
- CPU가 어떻게 시간을 알뜰하게 쓰면서 명령어들을 처리할까,
- 이것도 중요함

명령어 파이프라인

- 슈퍼 파이프라인 이라고도 부름
    - 책마다 다른것
- 명령어 병렬처리
- 실행하는 시간동안 각 "단계"만 안겹치게 하면
- CPU는 하나의 단위시간에 명령어 여러개를 동시에 실행할 수 있다
- 명령어가 처리되는 과정을 비슷한 시간 간격으로 나누면?
    1. 명령어 인출 instruction Fetch
    2. 명령어 해석 instruction Decode
    3. 명령어 실행 Execute instruction
    4. 결과 저장 write Back
    - 전공서에 따라
        - 인출->실행,
        - 해석->실행->접근->저장
        - 으로 나누기도 함

파이프라인 위험

- 명령어 파이프라인이 성능 향상에 실패하는 경우
- 병렬처리를 실패한다는 뜻

데이터 위험

- 명령어 간의 의존성에 의해 발생

- 모든 명령어를 동시에 처리할 수는 없다
- 이전 명령어를 끝까지 실행해야만 비로소 실행할 수 있는 경우
  - 명령어 1 : 31 = R1 + R2
    - R2 레지스터 값과 R3 레지스터 값을 더한 값을 RI 레지스터에 저장
  - 명령어 2 : R3 = R3 + R4
    - R1 레지스터 값과 R5 레지스터 값을 더한 값을 R4 레지스터에 저장
  - 비순차적 명령어 처리를 통해 이 문제를 해결하려고 노력함
  - CPU 입장과 code 입장이 다른것

제어 위험

- 프로그램 카운터의 갑작스러운 변화
- 그러면 프로그램 카운터에서 다음으로 예정된 다음 명령어가 무시됨
  - 병렬처리로 부분적으로 실행된 명령어는 완수되지 못한 헛수고가 됨
- 이걸 방지하는 기술이 있다
  - 분기예측 branch prediction

구조적 위험

- 서로 다른 명령어가 같은 CPU 부품(ALU, 레지스터)를 쓰려고 할 때

RAM

- Random Access Memory
- 임의 접근
- 직접 접근이라고도 부른다
- 현재 실행되는 명령어와 데이터를 저장
- 프로그램이 실행되려면 메모리에 저장되어 있어야 함

RAM의 용량과 성능

- RAM이 작아서 프로그램 ABC를 다 담을 수 없다면
  - 보조기억장치에서 가져오는 과정이 더 자주 발생해야 함

RAM의 종류

DRAM

- Dynamic RAM
- 전원이 연결되어 있어도 저장된 데이터가 동적으로 사라지는 RAM
- 데이터 소멸을 막기 위해 주기적으로 재활성화(refresh) 해야 함
- 일반적으로 메모리로 사용
  - **상대적으로**:
    - 소비전력이 낮음
    - 저렴함
    - 집적도가 높음 (부피 대비 저장량이 많음)
  - 대용량으로 설계하기 용이

SRAM

- Static RAM
- 전원이 연결된 동안 저장된 데이터가 정적으로 유지되는 RAM
- DRAM보다 일반적으로 입출력이 더 빠름
- 일반적으로 캐시 메모리에서 사용
  - **상대적으로**:
    - 소비전력이 높음
    - 가격이 높음
    - 집적도가 낮음
  - "대용량으로 설계할 필요는 없으나 빨라야 하는 장치"에 사용

DRAM vs SRAM

DRAM

- 재충전: 필요함
- 속도: 느림

- 가격: 저렴함
- 집적도: 높음
- 소비 전력: 적음
- 사용 용도: 주기억장치(RAM)

SRAM

- 재충전: 필요 없음
- 속도: 빠름
- 가격: 비쌈
- 집적도: 낮음
- 소비 전력: 높음
- 사용 용도: 캐시메모리

SDRAM

- "S"ynchronous DRAM
- 클럭 신호와 "동기화"된 DRAM
- 특별한 (발전된 형태의) DRAM

DDR SDRAM

- Double Data Rate SDRAM
- 대역폭을 넓혀 속도를 빠르게 만든 SDRAM
  - 대역폭은 데이터를 주고받는 길의 너비
  - 클럭당 데이터를 많이 받을 수 있음
- 특별한 (발전된 형태의) DRAM
  - 결국 DRAM의 발전 형태

DDR vs SDRAM

- **DDR SDRAM > SDRAM (Single Data Rate SDRAM)**

DDR2 SDRAM

- DDR SDRAM의 4배 대역폭

DDR3 SDRAM

- DDR SDRAM의 8배 대역폭

DDR4 SDRAM

- DDR SDRAM의 16배 대역폭
- 현대에 가장 대중적임

메모리에 바이트를 밀어 넣는 순서 - 빅 엔디안 / 리틀 엔디안

빅 엔디안

- 낮은 번지의 주소에 "상위" 바이트부터 저장
- 인간이 일정적으로 숫자체계를 읽고 쓰는 순서와 동일
    - 그래서 디버깅할 때 편함

리틀 엔디안

- 낮은 번지의 주소에 "하위" 바이트부터 저장
- 수치 계산이 편리함

캐시 메모리

- CPU와 메모리 사이에 위치한,
    - 레지스터보다 용량이 크고
    - 메모리보다 빠른
    - SRAM 기반의 저장 장치
- CPU의 연산 속도와 메모리 접근 속도의 차이를 조금이나마 줄이기 위해 탄생
- CPU가 매번 메모리에 왔다 갔다 하는 건 시간이 오래 걸리니,
    - 메모리에서 CPU가 사용할 일부 데이터를 미리 캐시 메모리로 가지고 와서 쓰자

계층적 캐시메모리

- L1-L2-L3 캐시
  - 일반적으로 L1 캐시와 L2 캐시는 코어 내부에 위치
  - L3 캐시는 외부에 위치
- 멀티코어 프로세서의 캐시메모리에서 L3는 코어 외부에 단일로 존재함
- **분리형 캐시**
  - {L1D, L1I} - L2 -(코어 외부로)- L3

참조 지역성의 원리

- 캐시메모리는 메모리보다 용량이 작다
- 당연하게도 메모리의 모든 내용을 저장할 수 없다
- CPU가 자주 사용할 법한 내용을 예측하여 저장

예측이 들어맞을 경우

- **캐시 히트**: CPU가 캐시 메모리에 저장된 값을 활용할 경우
- **OR 캐시미스**

캐시 적중률

- 캐시 히트 횟수 / (캐시 히트횟수 + 캐시 미스횟수)
- 캐시 적중률을 높여야 함
  - CPU가 사용할 법한 데이터를 잘 예측해야 함

참조 지역성의 원리 = CPU가 사용할 법한 데이터를 예측하는 방법

- CPU가 메모리에 접근할 때의 주된 경향을 바탕으로 만들어진 원리
  - CPU는 최근에 접근했던 메모리 공간에 다시 접근하려는 경향이 있다.
    - **시간 지역성 (temporal locality)**
  - CPU는 접근한 메모리 공간 근처를 접근하려는 경향이 있다.
    - **공간 지역성 (spatial locality)**

쓰기정책과 일관성

- **즉시쓰기**
  - 캐시 메모리와 메모리에 동시에 쓴다
  - 메모리를 항상 최신 상태로 유지
    - 둘 사이에 일관성이 깨지는 상황을 방지
    - 느림 = 버스 사용 시간 + 쓰기 시간
- **지연쓰기**
  - 캐시 메모리에만 값을 써 두었다가 추후 수정된 데이터를 한 번에 메모리에 반영
    - 더 빠르지만, 일관성이 깨질 수 있다는 위험을 내포

RAID

- **Redundant Array of Independent Disks**
- 하드 디스크와 SSD로 사용하는 기술
- 데이터의 안전성 혹은 높은 성능을 위해 여러 물리적 보조기억장치를 마치 하나의 논리적 보조기억장치처럼 사용하는 기술
- 1TB 하드 디스크 4개로 RAID를 구성하면 4TB 하드 디스크 1개의 성능과 안전성을 능가

RAID 레벨

- RAID를 구성하는 기술
- RAID 0, RAID 1, RAID 2, RAID 3, RAID 4, RAID 5, RAID 6

RAID 0

- 데이터를 단순히 나누어 저장하는 구성 방식
- **스트라입 (stripe)**:
  - 마치 줄무늬처럼 분산되어 저장된 데이터
- **스트라이핑 (striping)**:
  - 분산하여 저장하는 것

장점

- 입출력 속도의 향상
- HDD가 `n`개면 입출력의 주체가 `n`개니까 속도 상승

단점

- 저장된 정보가 안전하지 않음
- HDD 1개가 고장나면 데이터가 깨짐

RAID 1

- **미러링 (mirroring)**:
  - 복사본을 만드는 방식
  - 데이터를 쓸 때 원본/복사본 두 곳에 저장

장점

- 데이터 백업/복구가 쉬움

단점

- 복사본이 만들어지는 용량만큼 사용 불가
  - 많은 양의 HDD가 필요 (비용 증가)
- HDD 개수가 한정되었을 때 사용 가능한 용량이 적어짐
- 느린 쓰기 속도

RAID 4

- RAID 1처럼 완전한 복사본을 만드는 대신, 오류를 검출하고 복구하기 위한 정보(패리티 비트)를 저장
- **CS에서 패리티 비트**는 오류 검출만 가능하지만 RAID 4에서는 복구 가능

단점

- 패리티 디스크의 병목
  - HDD 123에 정보를 쓸 때마다 패리티 비트를 담는 디스크4도 접근해야 하니 병목 현상 발생

RAID 5

- 패리티 정보를 분산하여 저장하는 방식

RAID 4 vs RAID 5

- RAID 4: 패리티를 저장한 장치를 따로 둠
- RAID 5: 패리티를 각 HDD에 분산하여 저장

RAID 6

- 두 종류의 패리티(오류를 검출하고 복구할 수 있는 수단)
- RAID 5보다 안전하지만, 쓰기는 더 느림

요약

- 각 RAID 레벨마다 장단점이 있음
- 어떤 상황에서 무엇을 최우선으로 원하는지에 따라 최적의 RAID 레벨이 달라질 수 있음
- 각 RAID 레벨의 대략적인 구성과 특징을 아는 것이 중요

장치 컨트롤러

- **입출력 제어기 (I/O controller)**, **입출력 모듈 (I/O module)** 등으로 불림

역할

- CPU와 입출력장치 간의 통신 중개
  - 일종의 번역가 역할
- 오류 검출
- 데이터 버퍼링
  - **버퍼링**:
    - 전송률이 높은 장치와 낮은 장치 사이에 주고받는 데이터를
    - 버퍼라는 임시 저장 공간에 저장하여 전송률을 비슷하게 맞추는 방법

구조

- 버스에 연결되어서 입출력장치와 통신함

장치 드라이버

- 장치 컨트롤러의 동작을 감지하고 제어하는 프로그램
- 장치 드라이버가 설치되어 있지 않다면 해당 입출력장치를 사용할 수 없음

비교

- **장치 컨트롤러**: 입출력장치를 연결하기 위한 **하드웨어적인 통로**
- **장치 드라이버**: 입출력장치를 연결하기 위한 **소프트웨어적인 통로**

운영체제와의 관계

- 컴퓨터(운영체제)가 연결된 장치의 드라이버를 인식하고 실행할 수 있다면:
  - 컴퓨터 내부와 정보를 주고받을 수 있음
- 컴퓨터(운영체제)가 장치 드라이버를 인식하거나 실행할 수 없다면:
  - 그 장치는 컴퓨터 내부와 정보를 주고받을 수 없음

3가지 프로그램 입출력 방식

- **프로그램 입출력 / 인터럽트 기반 입출력 / DMA 입출력**

프로그램 입출력

- 프로그램 속 명령어로 입출력장치를 제어하는 방법
- 입출력 명령어로써 장치 컨트롤러와 상호작용

메모리에 저장된 정보를 하드 디스크에 백업하면 일어나는 일

1. CPU는 하드 디스크 컨트롤러의 제어 레지스터에 쓰기 명령 내보내기
2. 하드 디스크 컨트롤러는 하드 디스크 상태 확인 후 상태 레지스터에 준비완료 표시
3. CPU는 상태 레지스터를 주기적으로 읽어보며 하드 디스크의 준비 여부를 확인
   - 하드 디스크가 준비되었다면 백업할 메모리의 정보를 데이터 레지스터에 쓰기
   - 아직 백업 작업(쓰기 작업)이 끝나지 않았다면 1번부터 반복

        - 쓰기가 끝났다면 작업 종료
- CPU가 장치 컨트롤러의 레지스터 값을 읽고 쓰며 수행
- 예시:
    - 프린터 컨트롤러의 상태 레지스터를 읽어라
    - 프린터 컨트롤러의 데이터 레지스터에 100을 써라
    - 키보드 컨트롤러의 상태 레지스터를 읽어라
    - 하드 디스크 컨트롤러의 데이터 레지스터에 'a'를 써라

프로그램 입출력 방식

- CPU가 장치 컨트롤러의 레지스터 값을 알기 위한 방식은 두 가지
    1. **메모리 맵 입출력**
        - 메모리 주소 공간과 입출력 장치 주소 공간을 하나로 간주
        - 예:
            - 516번지: 프린터 컨트롤러의 데이터 레지스터
            - 517번지: 프린터 컨트롤러의 상태 레지스터
            - 518번지: 하드 디스크 컨트롤러의 데이터 레지스터
            - 519번지: 하드 디스크 컨트롤러의 상태 레지스터
        - 메모리 접근 명령어 = 입출력 장치 접근 명령어
    2. **고립형 입출력**
        - 메모리 주소 공간과 입출력 장치 주소 공간을 분리
        - 입출력 전용 명령어 사용

메모리 맵 입출력 vs 고립형 입출력

- **메모리 맵 입출력**
    - 메모리와 입출력 장치는 같은 주소 공간 사용
    - 메모리 주소 공간이 축소됨
    - 메모리와 입출력 장치에 같은 명령어 사용 가능
- **고립형 입출력**
    - 메모리와 입출력 장치는 분리된 주소 공간 사용

    - 메모리 주소 공간이 축소되지 않음
    - 입출력 전용 명령어 사용

인터럽트 기반 입출력

- 플래그 레지스터 속 인터럽트 비트가 활성화되면 인터럽트 발생
- 하드웨어 인터럽트는 입출력 장치가 아니라 장치 컨트롤러에 의해 발생

동시다발적인 인터럽트

- 입출력 장치가 많을 때 발생
- 모든 인터럽트를 순차적으로 처리하기 어려움
- **NMI (Non-maskable Interrupt)**:
    - 정전이나 고장 등 중요한 상황에서 발생
    - 다른 인터럽트를 처리 중이라도 무조건 처리

우선순위를 반영한 인터럽트

- **PIC (Programmable Interrupt Controller)**:
    - 장치 컨트롤러의 하드웨어 인터럽트 우선순위를 판단
    - NMI는 우선순위 판단 없이 CPU에 직접 전달

프로그램 입출력과 인터럽트 기반 입출력 공통점

- 입출력 장치와 메모리 간 데이터 이동은 CPU가 주도
- 이동하는 데이터는 반드시 CPU를 거침

DMA 입출력

- **Direct Memory Access**
- CPU를 거치지 않고 입출력 장치가 메모리에 직접 접근
- **DMA 컨트롤러**라는 하드웨어 사용

작동 과정

1. CPU가 DMA 컨트롤러에 입출력 작업 명령

2. DMA 컨트롤러는 CPU 대신 장치 컨트롤러와 상호작용하여 작업 수행
3. 작업 완료 후 DMA 컨트롤러가 인터럽트를 통해 CPU에 완료 알림
4. CPU는 입출력 작업의 시작과 끝만 관여

문제

- DMA 과정에서 시스템 버스를 사용
- 시스템 버스는 공용 자원이므로 동시 사용 불가
- **사이클 스틸링**:
  - CPU가 시스템 버스를 사용하지 않을 때 DMA 컨트롤러가 사용
  - CPU가 일시적으로 시스템 버스를 포기하고 DMA 컨트롤러에 사용 허가

입출력 버스

- 여러 장치 컨트롤러가 시스템 버스에 직접 연결되면 문제 발생
- **입출력 버스**:
  - 장치 컨트롤러는 입출력 버스를 사용하여 시스템 버스 이용 빈도를 낮춤
  - 구조:
    - `장치 → 입출력 버스 → DMA 컨트롤러 → 시스템 버스`

입출력 버스 종류

- PCI 버스, PCI Express (PCIe)
- 슬롯 → 입출력 버스 → 시스템 버스

GPU의 용도와 처리방식

용도

- 대량의 그래픽 연산
- 최근에는 CPU의 연산범위까지 확대되어 다양한 분야에 대한 연산이 가능

특징

- **코어의 갯수**:

    - 개별 코어의 성능은 CPU보다 떨어지지만 갯수가 월등하게 많음
    - **== 병렬처리에 용이함**
- 자체적 캐시메모리를 갖추고 있음

CPU를 대체할 수 없는 이유

- 개별 코어 성능이 낮기 때문
- 산술 연산과 같은 단순한 연산을 빠르게 병렬적으로 실행하는 데 적합