12\_09 시스템 아키텍처

141019 김연우

1. 프로세서 설계
   1. 프로세서의 실행단계
      1. Fetch
      2. Decode
      3. Execution
      4. Memory
      5. Write-Back
   2. 순차 동작
      1. 클록과 타이밍
         1. Clock : CPU가 하나의 명령어를 실행하는 단위
         2. 클락에는 전기가 통하는 때와 아닌 때로 구분
            1. 전기가 통할 때 : 조합회로를 통한 연산 수행
            2. 전기가 안통할 때 : 메모리 & 레지스터 & PC에 연산결과 저장
   3. 파이프라인 프로세서 설계
      1. 파이프라인 기본 원리
         1. 작업을 독립적인 세부작업으로 분할
         2. 각 단계에 순차적으로 작업 대상이 진입
         3. 여러 개의 작업 대상을 여러 단계에서 동시에 처리
      2. 프로세서에서 파이프라이닝
         1. 프로세서의 실행 단계를 단위로 작업을 분할
            1. 각 단계는 서로 독립적으로 작동 가능하기 때문
         2. 클락 별로 순차적으로 작업 대상이 실행 단위로 쪼개져서 진입
         3. 클락의 시간 단위를 더 작게 분해할 수 있다.
         4. 한 클락 마다 한 작업이 끝나는 것처럼 보이게 함.
      3. 파이프라인의 효과
         1. 시스템 처리율의 증가
            1. 단위 시간 동안 실행되는 명령어 개수
            2. 동시에 여러 개의 명령어가 실행되는 효과
         2. 명령어 응답시간 증가.
            1. 한 명령어의 실행완료 시간
            2. 느린 단계의 처리 시간 병목

비균일 분할의 문제

각 단계의 수행시간이 서로 다른 경우

가장 느린 단계에 의해 병목이 발생

* + - * 1. 단계들 사이의 레지스터 지연

실행 단계가 많아지면 처리량은 높일 수 있다.

하지만 레지스터 저장은 항상 해줘야한다.

단계가 너무 많아지면, 실제 실행 시간보다 레지스터 저장 시간이 커진다.

무한정 단계를 분해한다고 해도 처리량이 반드시 증가하지 않는다.

* + - 1. 파이프라인 해저드 (파이프 라인의 한계)
         1. 명령어들 사이의 의존관계에 의해 파이프라인이 진행할 수 없는 상황
         2. 종류

구조 해저드 : 하드웨어의 구조적 문제로 발생하는 것

동일한 하드웨어(메모리/레지스터)를 사용하는 단계의 중첩

FETCH & MEMORY

DECODE & WRITE-BACK

하드웨어 중복으로 해결 가능

I-cache, D-cache를 분리

듀얼 포트 구조

데이터 해저드 : 이전 명령어의 결과를 다음 명령어의 오퍼랜드로 사용할 때

READ-AFTER-WRITE 의 의존성

DECODE & WRITE-BACK

해결 방안

Stall : 그냥 지연한다. – 최악

Forwarding : 중간의 계산 결과를 레지스터로 전달

컴파일러 스캐쥴링 : 컴파일러에서 해저드가 발생하지 않도록 명령어 순서 재배치

이전 결과에 의존성이 있는 명령어를 뒤로 미루고 그렇지 않은 명령어를 중간에 배치

제어 해저드 : 점프문 등에서 기존 파이프라인의 연쇄가 멈추게 됨

PC값이 연결된 것이 아니므로 다음 연산을 미리 알 수 없음

해결 방법

분기 예측 : CPU가 다음 PC값을 예측

1. Program Performance :: 성능 이슈
   1. 프로그램에서 주요한 성능
      1. 시간 – 응답 시간
         1. 작업의 시작에서 종류까지 실제 경과 시간
         2. 한 프로그램을 위하여 사용된 CPU실행 시간
            1. 사용자 CPU 시간 + 시스템 CPU 시간
            2. 클럭당 시간 문제 : 하드웨어 아키텍처 (반도체)
            3. 한 명령어 당 사용 클럭 수 의 문제 : 마이크로 아키텍처
            4. 한 프로그램에 필요한 명령어 개수 문제 : 소프트웨어 아키텍처
         3. CPI (Cycle Per Instruction)
            1. 프로세서 아키텍처 설계의 효과적인 성능 기준.
            2. 파이프라인에서는 추가되는 버블의 비율만큼 증가.

분기 예측 정확도 증가하면 CPI 증가

* + - * 1. 평균 CPI 계산식

필요 인자 cpu time, 명령어 개수, 클럭 속도

Valgrind -–tool=exp-bbv : 실행 명령어 개수 파악할 수 있다.

Time : 실행 시간 파악할 수 있다.

cat /proc/cpuinfo : 클럭 속도 파악 가능

* + - * 1. 프로그램별 CPI가 다른 이유? (나의 추측)

프로그램 마다 명령어 개수와 종류가 다르다.

+@ 프로그램 로딩 언로딩에 필요한 운영체제 단위의 연산 로드가 있을 것이다.

* + 1. 비율 – 처리율
       1. MIPS (Million Instruction Per Second)
          1. 어떤 명령어를 실행할 것인가에 대한 이슈 (시프트 연산 vs 부동 소수점 연산)
          2. Dhrystone MIPS 가 자주 사용되는 벤치마크 MIPS 처리 명령어
       2. 벤치마크
          1. 성능을 측정하기 위해 선택된 프로그램 집합
          2. 성능 측정의 기준
       3. 암달의 법칙
          1. (소프트웨어) 시스템 성능 개선에 대한 기본 원칙
          2. 성능 개선된 각 프로그램이 전체 프로그램에서 차지하는 비율에 따라 전체 성능 개선효과를 계산한다.
          3. 암달의 법칙에 의한 성능 개선의 의미

자주 실행되는 부분의 최적화에 집중하라 (Common Case Optimization)

실행비중이 큰 부분을 식별하여 성능 개선에 활용

* + - 1. 프로파일링
         1. 프로그램의 함수 별로 실행 시간을 분석
         2. Gcc -01 –pg prog.c –o prog
         3. ./prog [args] 하면 gmon.out 파일 생성