HW-2

[1] DLX CPU 구조에서 수행 단계 (EX stage)에서 요구되는 모든 동작을 RTL (register transfer level) 표기로 기술하고, 해당 동작에 대한 의미를 주어진 파이프라인 구조를 고려하여 설명하세요.

ALU가 작동하는 단계로써, Operand에 수행하는 4가지 연산을 할 수 있도록 정의된다. 여기서 사용되는 Op code field는 ID/EX Pipeline register에서 다음 단계인 EX/Mem Pipeline register로 전달이 된다.

1. **Memory ref : ALUOutput <-- A + Imm**
   여기서 ALU의 역할은 Load, Store 두 가지 명령을 활용하여 필요한 effective memory address를 구하기 위해 사용된다.
   앞 단계에서 temporary register A에 읽어온 값과 displacement를 뜻하며 앞 단계에서 읽어온 Imm temporary register에 저장된 값을 더해서 얻게 되는 결과는 effective memory address가 되고, 이 결괏값을 저장하는 곳이 ALU temporary register (ALUOutput)이다.

2. **R-R ALU instr : ALUOutput <-- A op B**
   Op code field에 정의된 동작을 임시 레지스터에 이미 저장된 A와 B에 대한 ALU 연산을 수행하고 그 결과를 임시 레지스터인 ALUOutput에 저장하는 동작을 수행한다.

3. **R-I ALU instr : ALUOutput <-- A op Imm**
   여기서 Operand가 한쪽은 register (A), 다른 한쪽은 immediate data로 구성되어 있으며, 이 두 가지를 Op code에 명시된 대로 연산을 수행한 다음 그 결과를 임시 레지스터인 ALUOutput에 저장한다.

4. **C-BR : ALU1 <-- NPC + Imm & Cond <-- (A op 0)**
   Conditional Branch 명령어를 수행하는 경우에는, 어떤 조건이 충족되면 거기로 이동 (jump) 하는 것을 의미한다. 이는 PC relative jump를 수행한다고 한다. 따라서 만약에 컨디션이 충족된다고 하면 (Cond <-- (A op 0)) NPC 내용에다가 displacement를 더해서 (NPC + Imm), PC relative 한 address를 생성하고 그 결과를 ALU1에 저장한다. Pipeline 구조에서는 Multi-cycle 구조와 달리 한 개의 ALU를 추가하여 더 빨리 계산되도록 설계하였다.

[2] 파이프라인 구조에서 성능을 저하시키는 3 가지 요인 (Hazard)을 설명하고, DLX 구조에 서 각 Hazard를 해결/감소할 수 있는 방법을 요약 설명 하시오.

Hazard의 3가지 요인과 해결/감소 방법은 다음과 같다.

1. **Structural hazards** : Hardware 구조가 특정 명령어들의 조합을 Pipeline 내에서 수행할 수 없는 상황을 말한다. 예를 들어 Pipeline 구조 내에 한 개의 resource를 동시에 사용하려 할 경우 충돌이 생기는 상황이고, 이런 제한된 hardware resource conflict가 발생하는 상황을 structural hazard가 발생했다고 표현한다.
   이를 해결하기 위한 가장 심플한 방법은 stall 하는 것이다. 즉, 다음 명령어를 fetch 하는 게 아니라 stall(중지)하는 것이고 이를 위해 충돌이 발생할 수 있는 다음 명령어 전에 No Operation에 해당하는 단위 명령어인 bubble을 삽입하여 명령어 수행 완료를 고의로 delay 시킨다.
   또 다른 방법은 하나의 메모리에 동시에 접근이 가능하도록 dual port를 사용할 수도 있다.

2. **Data hazards** : Pipeline 내의 이전 명령어 수행 결과를 다음 명령어가 사용해야 할 경우, 앞의 명령어 수행이 끝나지 않았기에 다음 명령어 수행을 할 수 없는 상황을 말한다.
   이를 해결하기 위한 방법은 첫 번째로 stall 하는 방법이 있다. 이는 가장 간단한 방법이며, 지금 수행하는 명령어에서 사용되는 레지스터와 다음에 수행되는 명령어에서 사용되는 레지스터를 비교하여 그 값이 같을 경우, 현재 수행하는 명령어를 통해 원하는 레지스터 값이 나올 때까지 다음 명령어를 no operation 처리하여 (bubble을 삽입하여) stall 시키는 것이다.
   다른 방법으로는 Forwarding이 있다. Forwarding은 필요한 값이 준비되자마자 ID/EX 파이프라인 레지스터뿐만 아니라 다른 파이프라인 레지스터에서도 사용할 수 있도록 전달하는 것이며 이를 통해 ALU의 입력을 가져와 다음 명령어가 오류 없이 수행할 수 있도록 한다.
   하지만 이러한 forwarding 방법만으로 해결되지 않는 경우가 있는데, 이는 앞의 명령어가 load를 수행하고 뒤

에 명령어가 연산을 수행하는 경우이다. (Load latency 발생하는 경우) 이를 해결하기 위해서는 1 cycle stall을 하게 되는데 여기서 pipeline interlock 구조가 사용된다.
추가로 Data hazard가 하드웨어 요소로 해결이 안 된다고 하면 소프트웨어 요소로 해결 방법을 찾을 수도 있다. 다시 말해 수행해야 하는 일련의 명령어들을 수행하면서 stall이 불가피하게 발생할 경우, 명령어 수행 순서에 오류가 있지 않은 한, stall을 최소화하기 위해, 수행해야 하는 명령어 순서를 바꿔서 수행하도록 한다.

3. **Control hazards** : Branch 명령어와 다른 명령어가 Pipeline에 함께 진입해서 수행이 되는 경우 발생하는 문제이다. 예를 들어 Pipeline에 conditional jump 하는 Branch 명령어가 중간에 진입이 되어있으면 이 jump 명령어에 destination이 판단될 때까지 계속 fetch 되어 pipeline에 존재하는 다음 명령어를 무효화해야 하는 상황을 생기면 control hazard가 발생했다고 말한다.
이를 해결하려는 방법은 condition check을 앞에서 진행하는 방법이 하나 있고 (condition 여부를 판단하는 zero test를 ID/RF stage에서 실행), 이 condition에 따라 얻어지는 target address를 더 일찍 계산하는 방법이 있다 (Adder를 하나 더 써서 ID/RF 단계에서 NPC를 계산). 이렇게 하면 기존 3 cycle penalty를 1 cycle penalty로 줄일 수 있게 된다. 또한, 이 1 cycle delay slot을 활용하여 다른 작업을 수행하는 delayed branch를 통해 penalty를 없앨 수 있는 가능성도 존재한다.

[3] 메모리를 계층구조로 설계하는 이유/목적과 각 계층을 제어하는 주체는 무엇인지, Cache 메모리 설계에 활용되는 특성은 무엇이고, 어떠한 효과를 줄 수 있는 지 설명하세요.

메모리를 계층구조로 설계하는 이유/목적
메모리를 계층구조로 나누는 목적은 필요에 따라 메모리에 더 빨리 접근하여 Von Neumann Bottleneck 심화 현상을 완화하기 위함이다. 계층으로 나누면 지역성의 원칙을 (시간의 지역성, 공간적 지역성) 기반으로 효율성\적인 메모리 접근이 가능하기에 계층으로 나누지 않은 경우보다 빠르게 메모리에 접근할 수 있다. 다시 말해 컴퓨터는 속도가 느리고 용량이 큰 기억장치의 내용 중에서 CPU가 자주 사용하는 데이터를 속도가 빠르고 용량이 작은 기억장치로 옮겨 놓고 사용함으로써, 전체적인 기억장치 액세스 속도를 개선하는 전략을 사용할 수 있다.
또한, 경제적인 측면에서도 속도가 느려서 비교적 저렴한 기억장치부터 속도가 빨라서 비교적 비싼 기억장치까지 적절하게 배치한 계층구조를 통해 전체 메모리 시스템의 가격을 최소화하는 결과도 얻을 수 있다.

각 계층 제어 주체
메모리 계층구조의 가장 위에 있는 **Register**에서는 제어의 주체가 사용자이거나 software가 된다. 사용자가 해당하는 레지스터를 할당해서 쓰던지, 아니면 소프트웨어의 컴파일러 등이 레지스터에 필요한 데이터를 할당해서 활용하게 된다.
그다음에 위치한 **Cache**에서는 하드웨어적인 제어 방식으로 우리 눈에는 보이지 않지만, Cache memory controller라는 제어 주체가 자동으로 알아서 메인 메모리에서 Cache memory (high speed memory)에 load를 한 다음에 돌아가게 만든다.
그 아래 있는 **Main Memory와 Disk**는 둘 다 시스템 자원으로써 이를 효과적으로 관리하기 위해 Operating System이 제어 주체가 된다.

Cache 메모리 설계에 활용되는 특성 및 효과
Cache 메모리 설계는 지역성의 원칙을 기반으로 설계된다. 일반적인 프로그램에서 발생하는 수행 특성은 전체 프로그램 수행 시간의 90% 가 전체 코드 중 10%에 의해서 발생한다는 것이다. 그러면 이 10%에 해당하는 코드를 빠른 메모리 (즉 Cache 메모리)에 넣어 수행하게 되면 수행 시간의 90%에 해당하는 부분을 빠른 메모리에 할당해서 돌릴 수 있다. 이러면 프로그램의 대부분 수행이 빠른 메모리에서 돌아가는 것처럼 느낄 수 있는 효과를 주는 데이를 활용한 것이 캐시메모리 설계를 통해 얻을 수 있는 효과이다.
지역성의 원칙에는 크게 두 가지로 나뉜다. 하나는 시간적 지역성, 다른 하나는 공간적 지역성이다. 시간적 지역성은 CPU가 한 번 참조한 데이터는 다시 참조할 가능성이 높은 것을 의미하고, 공간적 지역성은 CPU가 참조한 데이터와 인접한 데이터 역시 참조될 가능성이 높다는 것을 말한다.