2장

2-1 ALU and data transfer instructions

#instruction set

컴퓨터가 제공하는 인터페이스로 이것을 이용해서 작업지시서를 만든다. 초창기에는 IS는 간단하게 구현하였다. 그러다가 메모리가 작아서 CISC가 유행을 했다. 메모리가 커지면서 RISC가 자리가 잡고, 후에 회사에서는 IS를 독점성을 위해 다르게 만들기 시작하였다. 하나의 프로세서가 있다고 가정할 때 사용자의 프로세서는 해당 응용프로그램에 맞게 돌아가게끔 만들어서 사용자는 그 응용프로그램만 사용하게끔 독점성을 이용한다. 하지만 이런 것들은 RISC기반으로 만들어진다.

질문) 범용 레지스터를 많이 만들면 좋지 않을까?

1. 32개의 범용 레지스터를 쓰는 것은 몇 번 레지스터를 쓸려면 5bit을 사용하게 되는데, 레지스터를 몇 개를 만드는 것은 opcode, operand의 길이에 연관되어 있다. 그래서 64, 32, 16 레지스터 중에서 어느 레지스터를 사용해야 좋다. 라는 질문에서는 대답을 할 수 없다. 어떻게 설계를 하여 벤치 마킹을 빨리 돌리는게 목표이기 때문이다. 레지스터의 개수보다는 성능이 중요하다.

2. 레지스터를 적게 사용하면 프로세서 입장에서는 레지스터에 있는 데이터를 접근하는 것이 빠르다. CPU가 메모리에 데이터를 접근하는 것은 수십억의 장소에서 하나를 찾는 것이다. 그래서 레지스터의 개수를 적게 써야 하나의 데이터를 찾고 읽고/쓰기 가 빨라진다. 즉 데이터 접근이 빨라진다

질문) load instruction은 뒤에 있는 레지스터에 왜 기준 레지스터와 상수 값을 주어지냐?

만약 32-bit address를 주게 되면 이미 instruction 자체가 32bit를 넘게 사용하기 때문에 공간을 두 개를 써야 된다. 그렇게 되면 접근을 두 번 해야 하기 때문에 성능이 떨어진다.

format을 두 개가 있는 게 무슨 의미가 있을까?

다른 format을 처리하는 하드웨어를 만들어 줘야 한다. 많은 format을 만들면 하드웨어는 해당하는 format을 만들어 줘야한다. 그렇게 되면 하드웨어는 복잡해진다. MIPS는 format이 3개 밖에 없어서 하드웨어가 단순하고 빠르게 만들어 줄 수 있다.

opcode을 몇 bit를 사용할 것인가 그리고 레지스터를 몇 개를 사용할 것인지 대한 해당사항들은 16-bit 상수 값을 설정한 것에 연관되어 있다.

물론 6-bit opcode, 16 register, 18-bit offset, 8-bit opcode, 32 register, 14 bit offset 등 후보 format을 생각할 수 있다. 이 후보들 중에서 벤치마킹을 돌려서 가장 성능이 좋은 것을 선택을 하는 것이다.

질문) RISC ISA는 어떻게 생겼나? 왜 그렇게 생겼나?

벤치 마킹 프로그램이 중요하다. 프로세서 기계는 소프트웨어를 빠르게 돌리는 것이다. ISA는 하드웨어 소프트웨어 instruction을 다루는 것이면, RISC은 자주 나오는 operation을 single operation으로 표현하는 게 목표이고 자주 나오지 않는 operation은 여러 개의 operation으로 표현되어도 괜찮다.

질문) 우리는 64-bit 프로세서에서 살고 있는데 왜 32-bit MIPS ISA를 배우고 있는가??

추가된 32-bit를 우리가 어떻게 사용해서 성능을 내는가?

결론은 MIPS 64-bit instruction set을 찾아보면 우리가 본 위에 그림이 나온다. 추가적인 32-bit을 사용하지 않는다. 레지스터 숫자도 늘리지 않는다. 똑 같은 32-bit instruction을 사용한다. 왜 그럴까?

1. MIPS 64는 MIPS 32에서 진화된 IS이다. MIPS 64을 많은 사람들을 쓰게 하기위해서 필요한 것만 추가해고 나머지들은 compatible해야 한다. 우리가 배운 것은 MIPS 32에서 핵심적인 내용이며, 이것 역시 MIPS 64에서도 핵심적인 내용이다.
2. 메모리 접근을 할 때 instruction 길이는 32-bit이기 때문에 두 개의 instruction을 한꺼번에 가져올 수 있는 것이다. instruction을 빠르게 가져오는 게 가장 중요하기 때문에 이 부분에서 굉장한 speed-up을 제공한다.

2-2 branch instructions

질문) I-format 에서는 offset 부분에서 16bit를 사용하는데 conditional branch에서 주소 값을 branch에 사용하는데 충분한가?

if문 컴파일할 때 jump bit은 얼마 들지 않는다. 벤치 마킹에서 jump distance bit을 이미 계산을 많이 했기 때문에 16-bit으로 충분하다.

질문) branch에서 사용하는 16-bit offset은 byte 단위인가?

보통 branch offset에서는 byte단위로 설정하지 않는다. word offset을 사용하는데 byte offset을 사용할 경우 하나의 instruction을 사용하는데 1word 즉 4byte로 사용하는데 그러면 bit에서 사용할 때 0과 2byte는 쓰지 않는 bit가 된다. 프로그래머들은 이런 낭비하는 bit를 선호하지 않기 때문에 word offset을 사용한다. 그러면 낭비했던 bit까지 사용하는 것이기 때문에 18-bit를 쓰는 효과가 일어난다.

질문) 그러면 offset을 사용하지 않고 R-format처럼 레지스터안에 offset을 저장하여 사용하지 않는가? (ex. beq r1, r2,r3)

RISC instruction에서는 branch에서 R-format을 사용하지 않는다. 만약 offset 값을 레지스터에 저장을 한다면 레지스터에 값을 넣는 instruction을 사용하기 때문에 결과적으로 2개 이상의 instruction을 사용하는 꼴이다. 그렇다면 성능에 대한 치명적인 단점이 생긴다.

질문) 주소에서는 32-bit가 필요하다. 만약 J-format을 사용하면 op에서 6bits와 나머지 destination address에서 26bits를 사용한다. 26bit를 word로 사용하기 때문에 28bit를 확보한 상태이다. 그러면 나머지 4bit는 어떻게 처리하는가?

reserved 공간에서 4bit을 사용하는데 이 부분에서 사용하지 않는 4bit이기 때문에 나머지 text 부분에서 28-bit만 텍스트에 있는 어디든 function을 표현할 수 있다.

질문) 비교연산을 따로 만들지 않고 branch 자체에서 비교연산을 만들면 되지 않는가? (ex. blt s1, s2, label)

equal, not equal 연산을 비교하는 것보다 less, more 비교 연산이 시간이 더 걸린다. 같다는 것을 비교하는 것은 bit by bit으로 병렬로 비교할 수가 있다. less than에서는 두 수를 빼고 더하기 연산을 sequence 하게 이루어진다. branch less than을 instruction을 하나로 만들어서 하나로 돌리게 되면 clock cycle time이 늘어나게 된다.

*pseudoinstructions(가상적인 머신 명령어): 가짜 instruction (어셈블러에서 사용함) move라는 instruction을 자주 사용하기 때문에 어셈블러 개발자의 편리함을 위해서 move연산을 따로 만들었지만 실제로는 zero 레지스터를 사용하여 add를 해준다.

2-3 Supporting program execution

프로그램이 실행시킨다는 것은?

프로그램을 실행시킨다는 것은 ISA 관점에서 machine instruction을 하나씩 실행시킨 것, 또는 fetch-decode-execute을 반복적으로 실행하는 것이다. HLL(high level language) 관점에서는 statement를 하나씩 실행시키는 것, 또는 function call return이 반복되는 것이다(procedure level).

질문) leaf procedures에서 4개 이상의 argument보다 필요한다면?

벤치 마크 프로그램 조사 결과 통상 argument의 사용 개수가 4개 이하였다. 그렇기 때문에 argument register를 4개로 지정한 것이다. 하지만 그 4개 수보다 많이 필요한다면 메모리에 접근하여 값을 할당하여 사용해야한다. 이럴 경우 속도는 줄어 든다.

질문) return value의 register은 왜 2개 필요한가?

c프로그램은 return 값을 1개만 지정한다. 하지만 double return value는 64-bit의 공간이 필요하기 때문에 double value를 return 할 때를 대비하여 레지스터 2개를 만들어 놓은 것이다.

질문) 하나 이상의 아이템을 return할 때에는 어떻게 하는가?

여러 개의 아이템을 return을 할 때에는 구조체로 묶어서 포인터로 주소 값을 return을 하든가, executable value을 사용하여 return을 한다.

질문) stack을 사용하는 이유는?

1. 프로세서 콜 될 때 stack에 push되고 프로세서가 끝나면 pop을 한다. 만약 stack으로 하지 않는다면, 많은 프로세서를 위해 저장 공간을 따로 만들어야 한다. 그러면 비효율적으로 저장 공간을 사용하게 된다. 스택을 사용하면 프로세서 콜 할 때, push, pop을 하기 때문에 그렇게 많은 공간을 만들 필요가 없어 진다. 그렇기 때문에 공간을 효율적으로 사용 가능하다.
2. recursion- 똑 같은 프로세서가 계속 반복적으로 부를 때(self-call)이 몇 번 일어나는지 모를 때 저장 공간을 미리 만들어 놓으면 해당 공간을 얼마나 차지할 지 예상을 못하기 때문에 stack을 사용한다.

static linking에서는 hello1.c ... 3개의 c 프로그래밍 코드가 라이브러리를 통해서 object를 생성하였는데 해당 object의 크기는 라이브러리와 합쳐서 만들어졌기 때문에 코드에 비해서 데이터가 커졌다. 하지만 dynamic linking에서는 라이브러리 shared object와 c 프로그래밍이랑 결합하지 않고 나중에 필요한 부분만 사용하기 위해서 linking을 한다. 그래서 상대적으로 execute 파일이 작다. 프로그램이 시작 또는 call로 인해서 불러졌을 때 linking으로 통해서 실행된다.

3.Computer arithmetic and ALU

overflow는 확실한 오류이다. 그렇다면 이런 문제는 프로그래머에서 해결을 해야 하는가? 아니면 하드웨어에서 해결을 해야 하는가?

<관점 – 프로그래머 해결> - c language

2의 보수에서 overflow가 나는 것은 전반적으로 프로그래머 잘못이기 때문에 충분한 크기의 데이터 타입을 사용해야 한다. 여기서 하드웨어는 절대로 개입하면 안된다.

addu, addiu, subu, 와 같은 instruction은 overflow를 무시하는데, 여기서 u는 unsigned를 가리키는 것이 아니라 2의 보수에 대한 오버프로우를 무시하라는 의미이다.

<관전 – 하드웨어 해결> - fortran

signed 연산에서 overflow가 나는 것은 백퍼센트 bug이기 때문에 하드웨어는 최소한의 bug를 잡

아서 stop을 시켜줘야 한다. add, addi, sub와 같은 instruction을 사용하는데 오버프로우를 예상 또는 발견하는 역할을 한다.

Floating Point Standard

1980년도에 제작이 된 표준 표현법은, 컴퓨터가 처음으로 네트워킹이 되던 시절에 floating point를 표현 방식을 컴퓨터 회사간에 독자적으로 표현을 하였기 때문에 호환성이 떨어졌다. 그래서 floating point standard가 생겼다. 그리고 754 standard는 1985년에 두개의 표현방식을 정의하였는데 single precision과 double precision입니다. C언어에서의 float가 single precision, double이 double precision을 의미합니다.

single-precision과 double-precision 두 가지를 만든 이유는 single-precision으로 표현이 안된 수가 당연히 종종 나왔다. 하드웨어가 미약하였기 때문에 double을 사용하지 않은 이유도 있다. 하지만 하드웨어가 허용되면 일반적으로 double-precision을 사용하는 게 더 좋다. 오늘 날에 하드웨어는 범용 컴퓨터에서 사용할 때에는 double을 사용한다. float를 쓸 때에는 이유가 있어야 한다. 예를 들면 작고 쉬운 임베디드 연산을 사용할 때 사용하지만 그 외에는 double을 사용해야 한다.

Associativity

fraction bit에 영향이 가는데, FP으로 연산을 하면 메모리 저장될 때 반올림에서 올라가기 때문에 1보다 약간 큰 값으로 근사 계산으로 되어진다. 그렇기 때문에 1과 같지 않아진다.

질문) 프로세서들은 multimedia을 어떻게 효과적으로 처리를 할까?

640bit adder가 있다 가정하에 16-bit carry라인을 끝어 버리면 4개의 독립적인 16-bit adder을 사용한다. 16-bit multimedia 데이터를 4개를 64-bit adder로 저장하고 이것을 4개를 병렬로 연산할 수 있다. 4개의 multimedia를 데이터를 동시에 연산할 수 있기 때문에 4배속으로 할 수 있다

4-1 Single Cycle Design

building blocks for r-types

해당 instruction을 예로 처음에 두 개의 레지스터를 읽을 수 있는 channel이 필요하다. 해당 channel은 32-to-1 mux을 사용하며, 32개의 레지스터 중에 하나의 레지스터를 선택을 한다. 더 깊게 들어가면 레지스터를 선택하기 전에 5-to-32 decoder을 통해서 신호에 따른 결과 값이 하나씩만 있기 때문에 5-to-32 decoder는 5-bit으로 표현되는 레지스터에서 신호를 받은 해당 레지스터만 mux으로 신호를 보낸다. 두 개의 레지스터를 읽으면 Read data 1, 2에 ALU 연산으로 보내는데

ALU연산에서도 마찬가지 어떤 연산인지 확인하기 위해 select operation와 zero signal이 존재한다. select operation에서 add, sub, 등의 signal이 opcode에 따라 결정되며 zero signal은 beq에서 사용되는데 ALU에서 빼기 연산으로 zero의 값이 나오면 jump를 한다. 현재 add의 연산이기 때문에 결과값을 다시 보내기 위해 ALU result를 통해 Write Data에 데이터를 보내고 Write register에 해당한 레지스터에 값을 저장한다. add라는 instruction을 실행시키면 두 개의 레지스터를 더해서 레지스터에 저장을 해야 하는데 여기서 RegWrite 신호를 enable를 해줘야 한다. 반대로 beq는 두 개의 레지스터를 같은 지 아닌지 비교만 하기 때문에 신호를 disable 즉, 0의 값을 줘야 한다. 이것도 역시 opcode에서 결정한다.

building blocks for lw/sw

load에서는 메모리를 읽기 때문에 MemRead 신호에 1을, store에서는 메모리에 쓰기를 해야 하기 때문에 MemWrite에 신호를 줘야 하고, 저장할 데이터도 받아야 한다. load/store에서는 immediate 값은 16-bit이기 때문에 ALU 연산에 보내기전에 Sign extend에서 32-bit으로 만들어 준다.

Branch Instructions

beq에서 immediate 값을 16-bit에서 32-bit으로 바꾸고 해당 immediate값은 word 단위이기 때문에 byte로 바꿔 주기 위해서 shift를 해준다. 두 개의 레지스터 같으면 ALU 연산에서 0의 값을 받아 Mux로 통해 해당 PC값을 jump를 하고 아니면 PC+4를 받는다.

RegDst에서 load는 I-format이기 때문에 결과 값을 쓰는 레지스터는 20-16이여야 한다. 그렇기 때문에 신호를 0을 준다. R-format일 경우 레지스터는 11-15을 쓴다. 신호를 1을 준다.

ALUSrc에서는 immediate의 연산을 사용하는지에 대한 신호이기 때문에 R-format은 레지스터간에 연산이기 때문에 0이라는 신호를 주고 load와 store은 각각 1을 준다.

MemtoReg에서는 R-format는 결과 값을 레지스터에 저장해야 하기 때문에 해당 MUX에서 아래 값을 0에 신호를 줘야 한다. 반대로 load에는 Readdata에 신호를 줘야하기 때문에 1의 신호를 받는다.

RegWrite는 R-format은 당연히 결과 값을 레지스터에 저장한다. 그렇기 때문에 1의 신호를 준다. (1 -> enable)

5-1

Physical memory model

- 가장 평범하고 단순한 모델

- CPU가 만드는 주소가 바로 메인 메모리의 주소가 되는 것

- 작은 임베디드 시스템에서 많이 사용한다.

- 어셈블리 프로그래밍을 할 때는 프로그래머가 메모리를 어떻게 사용할지 결정한다.

-> 사람이 메모리를 할당하기 때문에 복잡하게 만들지 않는다

- C 프로그래밍을 할 때는 컴파일러가 메모리를 어떻게 사용할지 결정한다.

- 1960년대 이전까지는 모든 컴퓨터가 이러한 방식을 사용했다.

VM가 생긴 이유

- 메모리와 주소가 1:1로 매칭되기 때문에 프로그램의 크기가 남아 있는 메모리 조각보다 크다면 기존에 프로그램의 위치를 옮겨주거나 프로그램을 나눠서 집어넣는 등 불편함이 존재했다.

- 성능을 내기 위해서는 프로그래머가 메인메모리의 크기에 따라 프로그램을 잘 설계해야했다.

-> 프로그래머가 신경 써야할 점이 많이 존재했다.

- 프로그램 컴파일 및 실행이 메인 메모리의 크기와 무관하게 하기 위해서 VM이 생겼다.

Virtual memory

- 프로그램마다 메인 메모리와 상관없는 가상의 메모리 공간을 만들어준다.

-> 이때 이 공간은 디스크에 만들어준다.

-> 하지만 디스크가 너무 느리기 때문에 실제로 CPU가 사용하는 공간은 메모리에 카피로 만들어 준다.

-> 이러한 행위를 캐싱(액세스 스피드를 높이기 위해서 가까운 곳에 카피를 만들어 주는 것)이라고한다

-> 메인 메모리의 크기는 캐싱을 해줄 수 있는 공간에만 영향을 주는 것이지 VM과는 상관없다.

-> 이때 메인 메모리도 CPU의 입장에서는 느리기 때문에 캐시 메모리를 사용한다 즉 2단계로 캐싱이 이루어 진다.

-> 이 두 가지 캐싱은 다른 한쪽이 존재하지 않아도 동작 할 수 있기 때문에 서로 독립적이다.

-> VM은 OS가 담당하고 캐시 메모리는 하드웨어로 구현되기 때문에 프로세서 디자이너가 담당한다.

- CPU가 디스크 상의 virtual address space의 주소(virtual address)를 넘겨주면 OS가 그것을 보고

메인 메모리에 캐싱이 되어있으면 그 위치에서 공급해준다.

Split vs Unified Memory

- 캐시를 명령어와 데이터로 구분하는 이유는 structural hazard를 피하기 위해서이다.

- 메인 메모리는 명령어와 데이터를 한번에 저장한다

-> 그 이유는 따로 저장하면 CPU의 사용 패턴에 따라 명령어를 많이 캐싱하거나 데이터를 많이 캐싱하는 행위가 가능해져서 성능이 올라가기 때문이다.

memory management

- 계층 구조를 가지는 메모리에서 CPU가 사용하려는 데이터는 계층의 위로 올리고 사용하지 않는 데이터는 계층의 아래로 내리는 것

- 범용 컴퓨터에서는 VM과 캐시 메모리에서 일어난다.

- 3개의 계층이 존재하기 때문에 캐시 매니지먼트와 버츄얼 매니지먼트가 존재한다.

- CPU가 필요로 하는 것이 캐시 메모리에 없다면 메인 메모리에서 올린다(이때 공간 지역성 때문에 블록 단위로 가지고 오는데 이것을 캐시 블록이라고 한다. 이때 디스크에서 메모리로 가지고 올때는 페이지 단위로 가지고 온다.)

-> 이러한 방식을 on-demand 방식이라고 한다.

-> 이때 하나를 가지고 오기 위해서는 기존에 있는 것을 하나 빼고 자리를 만들어줘야 한다.

- 기본적으로는 on-demand 방식이지만 오디오, 비디오 같은 멀티미디어 데이터는 순차적으로 오기 때문에 다음에 오는 데이터를 예측하는 prediction 방식을 사용한다

계층 구조로 메모리를 설계하는 이유

- 우리가 가지고 있는 메모리 기술이 이상적인 메모리 기술이 아니기 때문이다.

-> 이상적인 메모리 기술은 용량이 굉장히 크고 비용은 싸면서 속도가 빠른 기술이다.

-> 만약 이상적인 메모리 기술이 있다면 메모리가 계층구조를 가지지 않을 것이다

- 많은 데이터를 값싸고 크기가 큰 낮은 계층에 저장하고 필요한 데이터를 높은 계층으로 올림으로써 빠른 속도를 낼 수 있게 만든다.

-> 캐싱을 통해서 이상적인 메모리를 흉내내는 것이다.

Principle of Locality(굉장히 중요)

- 프로그램을 돌리면서 프로그램이 명령과 데이터를 액세스하는 것을 보면 모든 명령과 데이터를 같은 확률로 액세스 하는 것이아니다.

- 주어진 시간에서 보면 그 시간에 CPU가 주로 액세스하는 명령과 데이터가 존재한다.

-> 즉 CPU는 특정 시간에 특정 구역을 주로 접근 하는 것인데 이 특정 구역을 캐싱하여 캐시 메모리에 올려서 실행 속도를 높일 수 있다.

- 메모리가 계층 구조가 효율성을 가질 수 있게 만드는 원리이다.

- 시간 지역성과 공간 지역성으로 나뉜다

- 시간 지역성은 CPU가 한번 액세스 하면 곧 다시 액세스 하는 경향이 있다.

-> 대표적인 예시로 loop가 존재한다.

- 공간 지역성은 CPU가 한번 액세스 하면 그 주변을 다시 액세스 하는 경향이 있다.

Cache hit

- CPU가 필요한 데이터를 찾을 때 그 데이터가 캐시 메모리에 존재 할 때를 의미한다.

- 캐시 히트가 일어날 확률을 cache hit rate라고 하며 hits/accesses를 의미한다.

-> 이 확률이 95%가 될 수 있도록 설계한다.

Cache miss

- CPU가 필요한 데이터를 찾을 때 그 데이터가 캐시 메모리에 존재 하지 않을 때를 의미한다.

- 캐시 미스가 발생하면 메인 메모리에서 그 데이터를 포함하는 캐시 블록을 가지고 온다.

-> 이때 발생하는 시간을 miss penalty(중요함)라고 한다.

- 캐시 미스가 일어날 확률을 cache miss rate라고 하는데 1 - cache hit rate라고 한다.

Page hit

- 캐시 히트처럼 데이터를 찾을 때 그 데이터가 메모리에 존재 할 때를 의미한다.

-> CPU가 원하는 블록을 캐시로 올려준다.

Page miss(page fault)

- 캐시 미스처럼 데이터를 찾을 때 그 데이터가 메모리에 존재 하지 않을 때를 의미한다.

- OS가 개입을 해서 디스크에 존재하는 필요한 페이지를 가지고 온다.

Cache memory management

- 메인 메모리의 일부(캐시 블록)를 캐시 메모리로 캐싱 하는 것을 의미한다.

- CPU 속도에 맞춰야 하기 때문에 하드웨어로 구현되고 단순하며 빠르다.

- 프로세서의 일부이므로 프로세서 디자인에 포함된다.

-> 컴퓨터 아키텍쳐에서 다뤄야한다.

- 캐시 메모리의 존재를 OS와 소프트웨어는 알지 못한다.

Virtual memory management

- 디스크의 일부(페이지)를 메인 메모리로 캐싱 하는 것을 의미한다.

- 소프트웨어로 구현된다.

-> 속도를 위해서 하드웨어로 구현하면 비용이 많이 드는데 이미 많이 느리기 때문에 하드웨어로 구현한 효과를 많이 받지 못한다.

캐시 메모리의 입장에서 디스크를 고려하지 않는 이유

- page fault가 얼마나 자주 발생하는지 page fault가 발생할 때 얼마만큼의 시간이 드는지는 캐시 메모리 디자인과는 아무 상관이 없다.

-> 즉 캐시 메모리 디자인과 상관없는 page fault(메인 메모리에 데이터가 존재하지 않는 상황)를 빼버리기 때문에 디스크에 있는 모든 데이터가 메인 메모리에 전부 존재한다고 생각하는 것이다

Read hits

- CPU가 명령이나 데이터 읽으려고 했는데 캐시에 존재할 때를 의미한다.

Read misses (memory stalls)

- CPU가 명령이나 데이터 읽으려고 했는데 캐시에 존재하지 않을 때를 의미한다.

-> 메인 메모리에 가서 캐시 블록을 가지고 와야하는 시간인 미스 패널티만큼의 시간이 걸린다.

-> 이때 미스 패널티 동안 파이프라인은 동작 할 수 없기 때문에 stall이 생긴다.

Write hits

- 데이터를 쓰기를 할 때 쓰기 할 데이터가 캐시에 존재하여 캐시에 쓰기를 진행하는 것

-> 이때 캐시에 있는 데이터와 그 데이터의 원본인 메모리 사이에 값의 차이가 생긴다.

- 캐시에 데이터를 업데이트 할 때 메모리에 있는 데이터도 같이 업데이트 해주는 방식을 wirte-through라고 한다.

- 캐시에만 업데이트를 하고 메모리에 있는 데이터는 나중에 업데이트 해주는 방식을 withe-back 이라고 한다.

Write misses (memory stalls)

- 데이터 쓰기를 할 때 쓰기 할 데이터가 캐시에 존재하지 않는 것을 의미한다.

- Read misses와 마찬가지로 stall이 걸린다.

Memory stalls

- 캐시에 데이터가 존재하지 않아서 발생하는 stall을 의미하며 기존의 stall보다 훨씬 많은 사이클이 소모되기 때문에 발생 확률이 낮아도 큰 영향을 준다.

Average memory access time

- Hit time(데이터가 캐시에 존재할 때 걸리는 시간) + miss rate(데이터가 캐시에 존재하지 않을 확률) * miss penalty

Cache memory의 중요한 이슈

1. placement issue (메인 메모리에서 가져온 캐시 블록을 어디에 놓을 것이냐?)

2. identification issue (CPU가 원하는 item이 캐시에 존재 하는지에 어떻게 판단할 것이냐?)

Direct map cache

1. placement

- 메인 메모리에 모든 블럭에 캐시에 들어갈 때 어디로 들어갈지 미리 지정해둔다.

-> 이때 들어가는 위치는 블럭의 메인 메모리에서의 위치에서 캐시 메모리의 크기로 나눈 나머지의 위치로 이동한다

- 캐시 메모리 입장에서는 각 위치 별로 들어올 수 있는 블럭이 정해져 있는 것이다.

-> 이때 각 위치 별로 들어올 수 있는 블럭들 중에서 하나만 각각의 위치에 저장된다.

2. identification

- CPU가 입력하는 주소를 tag(캐시 블록이 어떤 위치에서 온 캐시 블록인지 지정), index(캐시의 몇 번째 블럭인지 지정), byte offset(블럭에서 찾고자 하는 item이 몇 번째 byte인지 지정)으로 나눠 생각할 수 있다.

-> identification에서 byte offset은 내가 원하는 블록을 찾고 그 블록에서 몇 번째 바이트를 CPU에게 줄 것인지를 결정하는 것이기 때문에 사용하지 않는다. (placement와도 무관하다.)

-> index가 찾아봐야 하는 캐시 메모리의 위치를 결정해준다.

-> 캐시 메모리에 블록을 저장할 때 블록을 가지고 온 주소의 앞부분인 tag를 같이 저장하여 identification에서 지금 찾는 item이 캐시 메모리에 저장되어 있는지의 여부를 주소의 tag부분과 캐시 블록의 tag를 비교하여 확인한다.

-> 즉 CPU가 입력하는 주소의 index 부분을 보고 캐시 메모리의 원하는 블록의 위치로 가서 캐시블록의 tag와 주소의 tag를 비교하여 같으면 캐시 히트라고 하고 다르면 캐시 미스라고 판단한다.

-> 만약 캐시 히트라면 offset을 이용해서 캐시 블록 안에서 원하는 item을 꺼내는 것이다.

- 캐시 메모리의 가장 앞에 있는 비트는 저장되어 있는 데이터가 의미 있는 값인지를 확인하는 비트로 재부팅 되었을 때 랜덤 비트로 바뀌는 캐시 메모리의 값을 사용하는 것을 방지한다.

+ 만약 byte offset이 2비트라면 블록이 담고 있는 데이터가 4바이트라는 것을 의미하고 CPU는 word단위이므로 offset으로 나눌 필요없이 바로 data를 보낼 수 있다. (과거에 사용하던 캐시에는 이러한 경우가 있었다.)

+ 또한 byte offset에서 마지막 2비트는 word단위로 CPU한테 값을 보내기 때문에 사용하지 않는다. 마지막 2비트를 제외한 비트를 word offset으로 사용하면 된다.

Cache simulation

- 캐시의 size, mapping, block size 등과 같은 값을 미리 정해 놓고 CPU가 주소를 줬을 때 캐시 히트 비율을 판단하는 것

- 벤치마크를 돌릴 때 나오는 주소를 모은 것을 어드레스 트레이스라고 하는데 이는 굉장히 크다

-> 또한 속도가 굉장히 빨라서 소프트웨어로 기록할 수 없기에 특별한 하드웨어 툴이 필요하다

Cache performance

1. 캐시 메모리의 사이즈 확장

- hit time은 늘어나지만 miss rate가 줄어든다

-> 전체적인 average access time을 확인해야하며 실제로는 성능이 늘어난다.

2. 블럭 사이즈

- 블럭 사이즈를 바꾸는 것은 캐시 메모리의 사이즈와는 다르게 비용이 들지 않지만 성능에는 영향을 준다.

-> 중요하게 생각해야하는 파라미터이다.

- 블럭 사이즈가 너무 작으면 공간 지역성을 활용하지 못하기 때문에 성능이 떨어진다.

- 블럭 사이즈가 너무 크면 캐시안에 블럭이 많이 들어가지 못한다.

-> 이때 프로그램 실행 될 때 자주 쓰이는 영역(working set)이 캐시안에 들어갈 수 있는 블럭보다 많으면 미스가 많이 발생한다

-> 블럭 사이즈가 커지면 블럭을 옮기는데 시간이 더 걸리기 때문에 miss penalty도 커진다

- 캐시 시뮬레이션으로 최적의 블럭 사이즈인 sweet spots exist를 찾을 수 있다.

3. wide bus

- 버스 폭을 늘리면 전송 속도가 빨라져서 miss penalty가 줄어든다.

4. interleaved memory

- 메모리를 구성할 때 한 덩어리로 구성하는 것이 아니라 독립적인 덩어리(memory bank) 여러 개로 만드는 것이다.

- 캐시 미스가 발생하면 매 사이클마다 하나의 신호를 각각의 독립적인 뱅크에 보내서 뱅크가 독립적으로 동작하게 만들어 전체적인 처리 속도를 증가 시켜서 miss penalty를 줄인다.

cache hit rate

- 데이터보다 명령이 지역성이 더 잘나와서 I-cache가 D-cache보다 hit가 더 잘된다.

-> 파이프라인에서는 매 사이클마다 fetch를 진행하기 때문에 I-cache는 매 사이클마다 접근한다.

-> 하지만 D-cache는 load/store에서만 접근하기 때문에 평균 hit rate를 낼 때는 I-cache의 hit rate를 더 많이 반영한다.

Direct mapping

- placement, identification, replacement는 이미 각 블럭이 캐시로 들어갈 때 위치가 고정 되어있기 때문에 아주 쉽다.

-> 아주 간단하기 때문에 빠른 동작으로 이어진다.

- address conflict 발생하여 miss rate가 증가할 수 있다.

Address conflict

- 주어진 순간에 자주 쓰이는 블럭들이 같은 위치로 매핑 될 수 있다.

-> 급격하게 miss rate가 증가하고 프로그램의 수행 속도가 크게 감소한다.

- 다른 코드를 수정하였을 때 갑자기 address conflict가 발생할 수 있다

-> 이때 프로그램의 속도가 떨어졌음에도 캐시의 address conflict 때문에 발생한 문제라고 인식하기가 힘들다.

- 캐시가 작으면 더 자주 나타난다.

-> 오늘날에는 캐시 메모리가 과거보다 커졌기 때문에 비교적 발생 빈도가 줄어들었지만 경우에 따라서 캐시메모리를 작게 만들어야 하는 경우에는 신경 써줘야 한다.

Two-Way Set-Associative Cache

- direct map 방식에서 캐시 메모리의 배열을 2열 종대로 바꾼 것이고 행을 set, 열을 way라고 한다.

- direct map 방식에서 발생하는 address conflict(2-way)를 해결하여 miss rate를 감소시킬 수 있다.

- 하드웨어가 조금 늘어나 hit time이 2%정도 증가한다.

-> hit time이 프로세서의 clock cycle을 결정하는데 속도가 조금 줄어드는 것이다.

1. placement

- direct map과 비슷하게 각 블럭이 캐시로 들어갈 때 들어갈 수 있는 set이 정해져 있다.

-> 이때 set에서 존재하는 두 개의 way 중에서 선택해서 들어간다.

-> 선택을 통해서 들어가기 때문에 2-way conflict가 해소된다.

2. identification

- 각 블럭이 캐시에서 들어갈 수 있는 set이 정해져 있지만 2개의 way가 존재하기 때문에 2개를 다 찾아봐야 한다.

-> 찾는 건 병렬로 진행하기 때문에 괜찮지만 하드웨어가 늘어나게 된다.

3. replacement

- direct map 방식에서는 placement가 곧 replacement였기 때문에 따로 replacement 방식이 필요하지 않았지만 이번에는 2가지 선택지가 존재하기 때문에 따로 정책이 필요하다.

- reference bit를 이용하는데 블럭이 참조되면 reference bit를 1로 바꾸면서 같은 set의 다른 블럭의 reference bit를 0으로 바꾼다.

-> 선택의 폭이 2개이므로 LRU 구현이 간편하다. (만약 LRU 구현이 복잡하면 근사적인 방법을 사용한다.)

- 교체해야한다면 최근에 참조되지 않은 reference bit가 0인 블럭을 교체한다.

Write-through

- 캐시에 data를 업데이트를 하면 메인 메모리와 값과 다른 상황이 생기므로 이를 막기위해 캐시가 업데이트 될 때 메인 메모리도 같이 업데이트를 해주는 방식이다.

- 캐시에 쓰는 속도는 빠르지만 메인 메모리에 값을 쓰는 행위가 시간이 많이 들기 때문에 그대로 쓰는 것이 아니라 write buffer를 사용한다.

-> 메인 메모리에 값을 쓰는 대신에 write buffer에 값을 쓰는데 이것은 캐시와 병렬로 일어나기 때문에 시간 손실이 없다.

-> 이후 write buffer가 해당 주소에 해당 값을 업데이트 해준다. (일종의 백그라운드 작업이다.)

Write-back

- data를 쓸 때 캐시에만 data를 업데이트 해주고 메인 메모리는 나중에 업데이트 해주는 방식

- dirty bit를 추가하여 캐시에 업데이트를 할 때 1로 바꿔준다.

-> 이것은 캐시에 업데이트가 일어났지만 메인 메모리에는 값이 업데이트 되지 않았다는 뜻이다.

-> dirty bit가 0이면 clean하다고 말한다.

-> 메인 메모리에 업데이트를 해주는 시점은 블럭이 다른 블럭에 의해서 교체될 때이다.

-> Write-through처럼 write buffer에 쓰고 백그라운드 작업을 통해서 실제 업데이트가 일어난다.

Write-through vs Write-back

- Write-back은 메인 메모리에 대한 업데이트를 한번에 모아서 할 수 있다는 장점이 있다.

-> 워드 단위로 전송하는 것이 아니라 블럭 단위로 전송하기 때문에 효율이 높다

- Write-through는 개념적으로 단순하고 구현하기가 쉽다.

-> 미스가 났을 때 Write-back는 dirty bit를 확인해줘야 하는데 Write-through는 확인할 필요가 없다.

4, 8, 16-Way Set-Associative Cache

- 각각 n-way conflict를 해소하여 miss rate를 감소시킬 수 있다.

- 각각 하드웨어가 조금씩 복잡해지기 때문에 hit time 조금씩 늘어난다.

- 만약 최대한의 자유도를 준다면 Fully-Associative Mapping이라고 한다.

-> 몇 way인지 정해져 있는 것이 아니고 캐시 블럭의 크기에 따라서 다르다.

-> index로 찾아 가는 것이 아니라 CPU가 보내준 Content를 패턴 매칭을 통해서 찾아낸다.

-> Content addressable memory라고도 한다.

-> 대표적인 예시로 TLB를 들 수 있다.

Multilevel Caches

- 캐시가 2개 이상의 계층을 차지하는 것을 멀티레벨 캐시라고 한다.

- CPU가 계속해서 빨라지다 보니깐 캐시 히트가 일어나도 CPU에 입장에서는 느리게 되었다.

-> 더 작고 더 빠른 캐시를 추가로 사용하게 되었다.

- CPU에 가까운 캐시를 L1 캐시라고 하고 그 다음 레벨 캐시를 L2 캐시라고 한다.

- clock 속도와 직접적으로 연관이 있는 L1 캐시는 hit time이 짧게 만드는 것을 목적으로 설계 했기 때문에 크기를 함부로 키울 수 없다.

-> 캐시는 크기가 커지면 속도가 자연스럽게 떨어진다.

- clock 속도와 직접적으로 연관이 없는 L2 캐시는 캐시를 충분히 크게 만들 수 있는 등 선택의 폭이 넓다.

+캐시는 로컬리티가 높을 때 좋은 성능을 보여준다. 만약 로컬리티가 좋지 않을 때는 캐시의 성능이 떨어지고 컴퓨터의 성능이 전체적으로 떨어진다.

+이론상 성능이 우수할 것이라고 생각 되는 알고리즘도 로컬리티를 고려하지 않으면 캐시 미스가 높아져 성능이 낮을 수 있다.

-> 즉 로컬리티는 프로그램의 성능에 큰 영향을 준다.

Virtual Memory - Placement

- page fault가 발생 했을 때 디스크에서 보내준 페이지를 어디에 저장할지에 대한 문제

- 이때 피지컬 페이지의 어디든 저장할 수 있기 때문에 최대의 자유도를 가지고 있는 fully-

associative mapping을 사용한다고 할 수 있다.

-> 캐시에서는 하드웨어로 구현되었기 때문에 fully가 복잡하다고 생각할 수 있는데 virtual memory는 소프트웨어로 구현되고 페이지 테이블로 관리하면 되기 때문에 쉽게 구현할 수 있다.

- 메인 메모리는 공유하지만 각 프로세스마다 고유의 페이지 테이블을 이용해서 공유하기 때문에 다른 프로그램의 페이지를 침범하지 않는다.

+ 페이지 테이블, PC, 레지스터는 프로세스의 중요한 상태 정보이므로 스위치가 일어날 때 교체해 줘야 하는 정보들이다.

-> OS는 각 프로세스마다 이것을 관리해준다.

Address Translation

- CPU가 내는 주소는 디스크에 있는 virtual address의 주소인데 실제로 디스크에 접근하면 속도가 굉장히 느리기 때문에 OS가 페이지 테이블을 이용해서 메인 메모리의 주소(physical address)로 바꾼다.

-> virtual address에 있는 페이지 넘버를 이용해서 페이지 테이블에서 메인 메모리에 캐싱이 되어 있나 확인하고 캐싱이 되어있다면 physical page number를 확인하여 메인 메모리에서 가져올 수 있는 것이고 이것을 페이지 히트라고 한다.

-> 만약 캐싱이 되어있지 않다면 page fault가 발생하는 것이고 디스크에 가서 페이지를 가져와 메인 메모리에 저장하고 페이지 테이블에 기록해준다.

Page Fault Penalty

- 만약 page fault가 발생하면 디스크에서 페이지를 가지고 오는데 이때 수백만 사이클이 걸린다.

-> 너무 많은 사이클이 소요되기 때문에 context switch(수천 사이클이 소모된다.)를 통해서 다른 프로그램을 실행한다.

- page fault가 일어나면 손실이 크기 때문에 page를 크게 만들거나(공간 지역성을 최대한 활용) fully associative placement를 사용하거나 스마트한 페이지 교체 알고리즘을 사용하는 등의 노력을 한다.

Write and Replacement

- disk write는 수백만 사이클이 소모 되기 때문에 write through는 사용할 수 없다

-> write-back 방식을 사용한다. (dirty bit를 사용한다.)

-> 디스크는 블럭 단위에 최적화 되어있고 워낙 느리기 때문에 워드 단위로 하나씩 쓰는 것이 아니라 한번에 많은 데이터를 쓰는 것이 유리하다.

- replacement는 시간 지역성에 충실한 LRU 방식을 사용한다.

-> 페이지 숫자는 굉장히 많기 때문에 근사적 LRU 방식인 reference bit를 사용하는 방식을 쓴다.

Fast Translation Using a TLB

- 페이지 테이블을 이용하기 위해서는 메인 메모리를 추가로 접근해야 하는데 많은 시간을 소모하는 행위이다.

-> 그리고 한번 CPU가 하나의 페이지안에 존재하는 데이터(or 명령)을 접근하면 로컬리티 때문에 한동안 그 페이지안에 있는 데이터들을 반복적으로 접근한다.

-> 같은 페이지 내부를 접근하는 동안은 똑같은 VPN을 내고 이것이 똑같은 PPN 변환되는 과정이 반복된다.

-> 그렇기 때문에 TLB라는 캐시에 이러한 VPN-PPN 페어를 저장하여 빠르게 변환한다.

TLB and Cache Interaction

- 기본적인 방식은 TLB를 이용해서 physical address로 변환하고 이를 이용해서 캐시에 데이터가 존재하는지 확인하는 방식인데 이렇게 하지말고 캐시가 virtual address를 직접 이용하여 캐시 히트 속도를 빠르게 한다.

-> 이것이 virtual cache 라고 한다.

-> 하지만 virtual address는 모든 프로세스가 공통으로 같은 주소를 가지고 있기 때문에 구분할 수 가 없다.

-> 캐시에서 데이터를 찾을 때 어떤 프로세스의 주소인지를 따로 처리를 해줘야하는데 이때 생기는 오버헤드로 인하여 실제로 성능이 증가하지 않아 사용하지 않는 방식이다.