컴퓨터구조(공영호)

Lee Jun Hyeok (wnsx0000@gmail.com)

February 13, 2025

목차

Part I

서론

1. 서론

1.1. 서론

1.1.1. 컴퓨터 구조

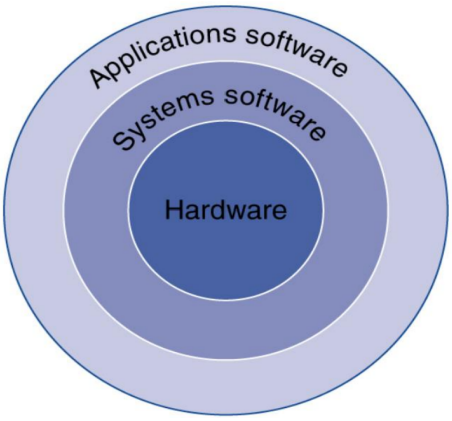
Definition 1 *컴퓨팅 플랫폼(하드웨어, 인터페이스, 시스템 소프트웨어 등)을 디자인하는 과학/예술을 컴퓨터 구조(Computer Architecture)라고 함.*

즉, 성능적/디자인적 목표를 위해 컴퓨터를 더 효율적으로 설계하는 것.

1.1.2. 실행 소프트웨어의 기저

고급 언어를 사용하여 제작한 소프트웨어의 밑에는 컴파일러, 운영체제 등의 시스템 소프트웨어가 있음. 또 그 밑에는 프로세서, 메모리, I/O 컨트롤러 등을 포함하는 하드웨어가 있음.

추상화되어 있는 고급 지식들은 복잡한 저급 지식을 몰라도 프로그래밍을 할 수 있도록 하지만, 더 나은 소프트웨어를 만들려면 저급 지식을 알 필요가 있음. 그렇기에 컴퓨터 구조를 배워야 함.

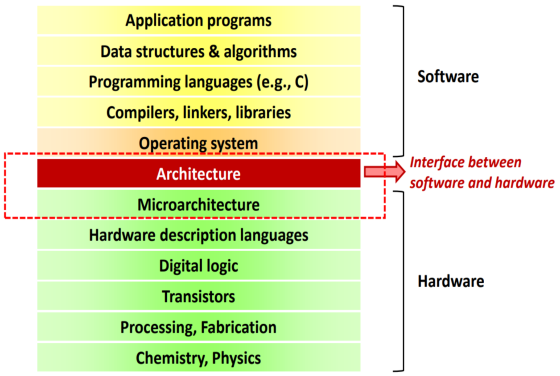


1.1.3. 학습 내용 개요

본 수업에서는 소프트웨어와 하드웨어 사이의 부분을 어떤 식으로 효율적으로 설계할 것인지를 학습함.

또한, 소프트웨어/하드웨어 개발자 모두에게 사용되는 어셈블리어 같은 인터페이스를 어떻게 디자인할 것인지, 주어진 공간에 트렌지스터를 어떻게 배치할 것인지, 메모리를 어떻게 사용할 것인지 등에 대한 학습을 함.

	Abstraction Levels	Examples
	Application Software	Programs
	Operating Systems	Device drivers
This Course	Architecture	Instructions, Registers
This Course	Micro architecture	Datapath, Controllers
This Course	Logic	Adders, Memories
This Course	Digital Circuits	AND gates, NOT gates
	Analog Circuits	Amplifiers
	Devices	Transistors, Diodes
	Physics	Electrons



1.2. 컴퓨터

1.2.1. 컴퓨터의 종류

컴퓨터는 PC(Personal Computer), 슈퍼컴퓨터, 서버컴퓨터, 임베디드 컴퓨터 등으로 나눌 수 있음.

PC는 다양한 소프트웨어를 구동하기 위해 일반적으로 사용되는 컴퓨터로, 비용/성능 면에서의 균형이 잘 조율되어 있음.

슈퍼컴퓨터는 과학/수학적 계산을 위한 하이엔드 컴퓨터로, 높은 성능을 가지고 있지만 전반적 컴퓨터 시장에서의 입지는 작음.

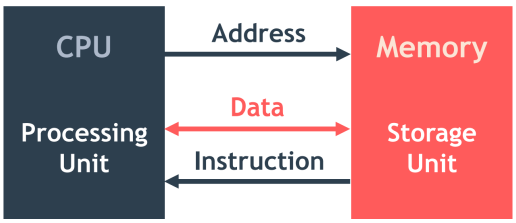
서버컴퓨터는 서버, 데이터센터로서 사용되는 컴퓨터로, 높은 성능과 안정성을 가지고 있고 그 규모가 큼.

임베디드 컴퓨터는 자동차, 기계, 가전제품 등에 들어가는 컴퓨터로, 엄격한 성능/비용적 제한을 만족시킴.

1.2.2. 폰 노이만 아키텍처

Definition 2 *폰 노이만 아키텍처(Von Neumann Architecture)는 최조의 컴퓨터 구조이면서 현대 컴퓨터의 기본적인 형태를 결정한 아키텍처임.*

폰 노이만 아키텍처는 cpu-memory의 구성을 가짐. cpu가 메모리에 주소를 주면 메모리에서 해당 주소에 있는 명령어를 cpu로 전달하고 cpu는 명령을 전달된 명령을 수행함.



1.2.3. 컴퓨터의 물리적 구조

폰 노이만 아케텍처를 따르는 컴퓨터들은 기본적으로 CPU(Central Processing Unit), Memory, I/O들로 구성되어 있음. 또한 CPU는 control과 datapath로 구성됨.

본 수업에서는 CPU, Memeory, 그리고 I/O 중 저장장치인 HDD와 SSD에 대해 공부함.

1.2.4. 마이크로프로세서

Definition 3 *마이크로프로세서(Microprocessor)는 작은 반도체 칩으로 만들어진 프로세싱 유닛으로, 컴퓨터 프로그램의 연산을 수행하기 위한 논리적 전기 회로를 가지고 있음.*

n비트 프로세서라고 할 때의 이 비트는 한 번에 처리할 수 있는 데이터의 처리량, 레지스터의 크기를 의미함. cpu는 4비트, 8비트, 16비트, 32비트, 64비트 순으로 발전되어 왔음. 32비트/64비트 운영체제는

32비트/64비트 프로세서가 있기에 존재할 수 있는 것.

마이크로프로세서는 cpu, gpu, npu 등을 포괄하는 개념임. cpu는 컴퓨터의 메인 처리 마이크로프로세서이고, gpu는 그래픽 처리에 특화된 마이크로프로세서, npu는 gpu에서 그래픽 처리 기능을 제외하고 ai 개발에 특화된 마이크로프로세서임. 참고로, gpu와 npu는 모두 cpu로부터 만들어졌음.

각 마이크로프로세서별로 사용하는 언어와 작동 방식이 다름.

예를 들어, Intel x86, ARM, MIPS 등이 있음.

1.2.5. cisc와 risc

Definition 4 *1. cisc*
cisc(complex instruction set computer) 명령어 개수가 많고 복잡한 작업을 가지는 아키텍처임.

코드 길이가 짧지만 명령어 별로 수행하는 작업이 복잡함. 적은 수의 명령어로 작업을 수행할 수 있어 컴파일러 구현에는 유리함.

2. risc
risc(reduced instruction set computer)는 명령어 개수가 적고 비교적 단순한 작업을 가지는 아키텍처임.

risc는 register 기반 명령어들을 가짐. 각 명령어들은 고정된 크기를 가지고, 메모리 접근에는 load/store 명령어만을 사용함. 또한 flag를 사용하는 조건부 코드가 없음.

risc는 작업 수행 시에 더 많은 명령어를 수행해야 하고 그에 따라 더 시간이 오래 걸릴 수 있지만, 최적화, 낮은 전력 소모, 낮은 가격, 작고 빠른 하드웨어에서의 실행에 있어 유리하기 때문에 임베디드 등에서 자주 사용됨.

ARM, RISC-V, MIPS, IBM PowerPC 등에서 사용함.

하드웨어를 업그레이드하면 성능은 충분히 보장되기 때문에, cisc와 risc의 선택에는 코드 호환성이 주된 지표로 기능함.

실제로는 cisc와 risc의 특성을 잘 조합하여 만든 cpu들이 존재함.

1.3. 프로세서의 발전

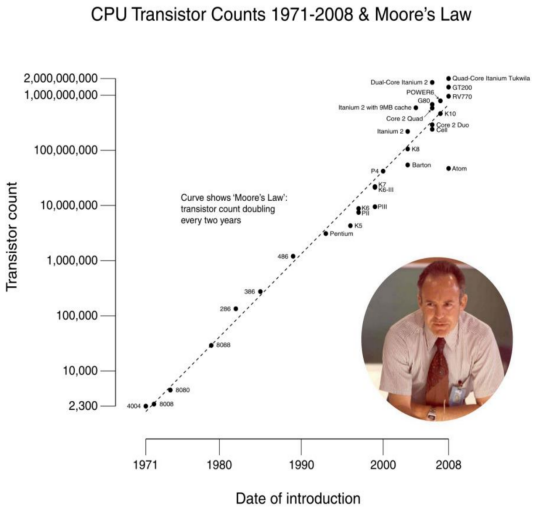
1.3.1. 무어의 법칙

Definition 5 *18 또는 24개월마다 반도체에 들어가는 단위 면적당 트렌지스터의 개수가 2배가 될 것이라는 것이 무어의 법칙(Moore's Law)으로 불리는 무어의 발언임.*

컴퓨터 구조에서 가장 유명한 법칙 중 하나.

cpu는 트랜지스터를 이용한 논리회로로 구성되어 있음. 트렌지스터가 많아지면 한 번에 처리할 수 있는 처리량이 늘어나고, 성능이 높아짐. 다만 이에 따라 발열이 심해짐.

이 법칙이 2000년대 초반까지는 나름 잘 들어맞다가, 단위면적당 전력소모량이 많아짐에 따라 발열이 심해지는 등의 한계에 봉착하여 현재는 그 증가폭이 더뎌진 상황.



1.3.2. 병렬화

Definition 6 *병렬화(Parallelism)를 통해 프로세서의 발전이 이루어졌음. 병렬화는 명령어 레벨의 병렬화(ILP, Instruction Level Parallelism), 데이터 레벨의 병렬화(DLP, Data Level Parallelism), 작업 레벨의 병렬화(TLP, Task Level Parallelism)로 나눌 수 있음.*

ILP는 명령어 수행에서의 병렬화, DLP는 하나의 명령어에 여러 개의 데이터를 실어서 처리하는 병렬화, TLP는 스레드를 늘리는 등 여러 작업을 동시에 수행하는 병렬화를 말함.

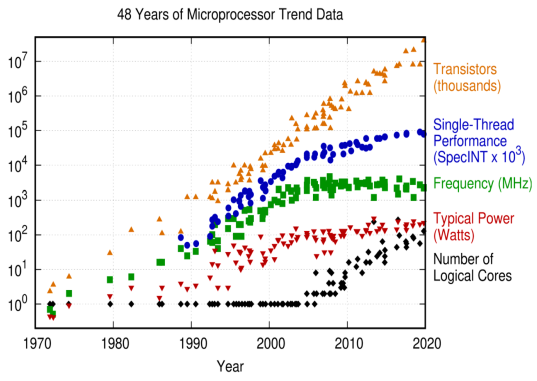
특히 DLP와 TLP는 gpu의 발전을 가속화시켰음. gpu는 많은 양의 데이터를 빠르게 처리할 수 있어야 하기 때문.

각 병렬화에 대한 내용은 뒤에서 자세히 다룰 예정.

1.3.3. 멀티코어로의 전환

트렌지스터의 개수가 늘어나며 단위면적당 전력소모량이 많아짐에 따라 발열이 심해져 파워의 한계(typical power)로 single-Thread의 성능 증가가 더뎌지는 지점을 마주하게 됨. 코어와 스레드를 여러 개 사용하는 멀티코어로의 전환은 이런 문제점의 해결책으로서 등장했음.

캐시 메모리 등도 증가했는데, 자세한 내용은 나중에 다룸.



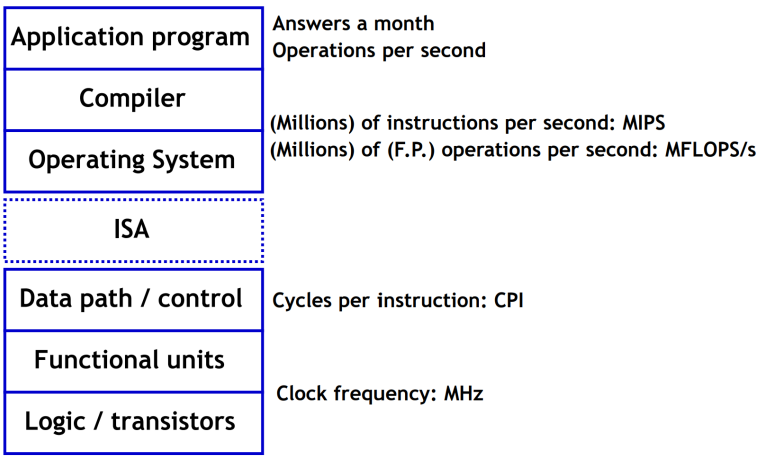
2. 성능

추후에 계산 연습도 하자.

2.1. 성능

2.1.1. 성능

컴퓨터에서의 성능은 프로세서 성능, HDD/SSD 성능, 메모리 크기, 그래픽, 전력 소모량, 가격, 무게 등으로 생각해 볼 수 있지만, 대부분의 성능 평가 기준은 **시간**임. 대체로 속도가 곧 성능인 것. 아래는 계층별 성능 척도임.



2.1.2. cpu 성능

Definition 7 *cpu의 성능은 실행 시간의 역수로 나타냄.*

```
Performance = 1 / Execution_time
```

즉, 시간이 줄어들면 성능이 좋아지는 것.

두 cpu 사이의 성능 차이는 Performance1 / Performance2로 계산이 가능함.

2.2. clock

2.2.1. clock

Definition 8 *cpu에서는 clock으로 시간을 측정함.*

cpu내에 있는 클록 펄스(clock pulse)를 기준으로 clock cycle time(클럭 주기)과 clock rate(클럭 속도, 클럭 주파수, clock frequency) 값을 계산하고, 이를 통해 시간을 측정하는 것.

클록 펄스는 수정 발진자(crystal oscillator) 등의 규칙적인 진동에 의해 발생하는 일정한 간격을 갖는 전자적 펄스를 말함. 이 펄스의 움직임을 그림으로 나타내면 아래와 같고, 신호가 올라가는 지점을 positive edge, 내려가는 지점을 negative edge라고 함.

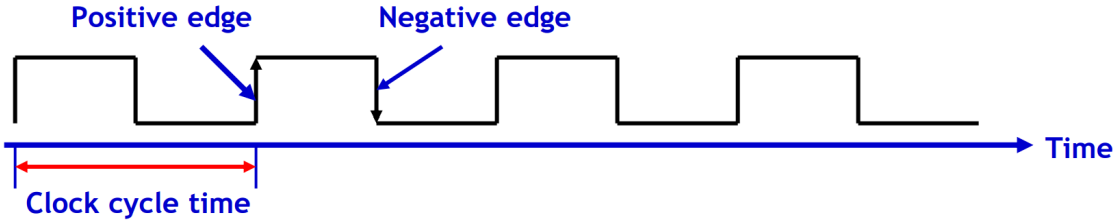
clock cycle time은 clock 신호가 0에서 1로 갔다가 다시 0으로 변하는 데 걸리는 시간임. (positive/negative edge에서 positive/negative edge 사이의 시간 간격.) 단위로는 ns(나노초) 등을 사용함.

clock rate는 clock cycle time의 역수로, 1초 동안 clock 신호가 몇 번 바뀌는지를 나타냄. 단위로는 Hz(헤르츠), MHz, Ghz, cycles/sec 등을 사용함.

즉, 컴퓨터에서는 clock으로 어떤 작업의 객관적인 시간을 측정함.

clock 신호는 아날로그 값으로부터 온 것으로, 실제로 각진 그래프의 데이터를 가지지는 않음. 지정한 기준점을 넘느냐 안 넘느냐로 그래프를 그린 것.

참고로, 기가, 메가, 킬로, 나노, 마이크로 등 물리학에서의 단위 체계는 필수 소양이라고 하심. 10의 지수가 12(테라, T),9(기가, G),6(메가, M),3(킬로, k), -3(밀리, m),-6(마이크로, μ),-9(나노, n)인 것들은 제대로 외우자.



2.2.2. 전파 지연 시간

Definition 9 *마이크로프로세서 내부 대부분의 회로는 하나의 clock 신호에 대해 동기화되는데, 이때 프로세서 내부에 있는 모든 회로가 다 동작할 수 있을 만큼 clock이 느려야 정상적으로 동작함. 이때 프로세서 내 회로 중 시간적으로 가장 먼 경로를 Critical Path라고 함.*

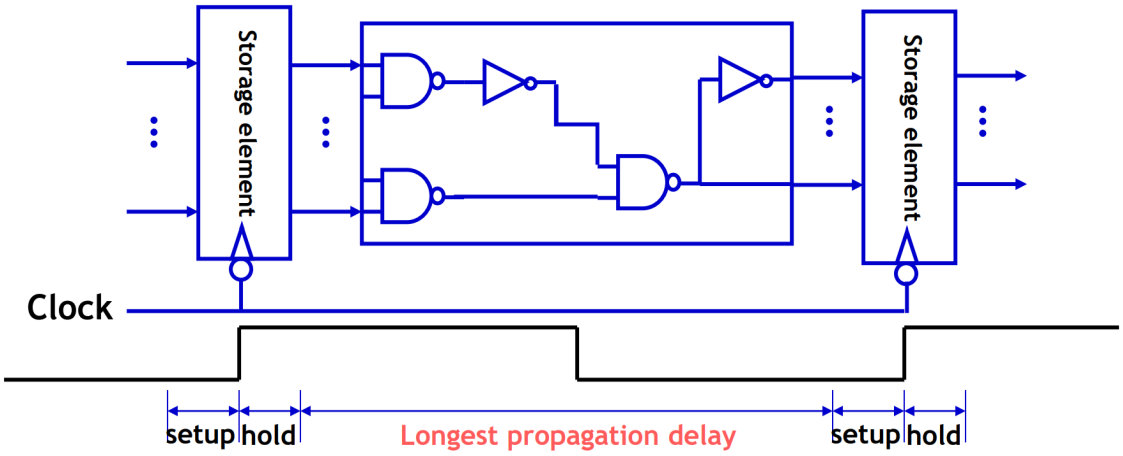
이때 Critical Path에서 걸리는 시간을 전파 지연 시간(Longest Propagation Delay, Critical Path Delay)라고 함.

마이크로프로세서의 clock cycle time의 지정은 프로세서의 회로와 연관이 있음. 프로세서에서는 clock에 적절히 맞추어 데이터를 송수신하기에, 각 장치들이 데이터를 안정적으로 주고받기 위해서는 동기화된 clock 신호를 가지고 있어야 함. 장치별로 성능에 따라 데이터를 수신받는 속도가 다르기 때문에, 시간적으로 가장 먼 거리까지 송수신이 될 정도로 clock이 느려야 하는 것임.

storage element 사이의 회로가 길고 복잡할수록 Critical Path Delay가 늘어남.

아래의 그림에서 setup time과 hold time은 1과 0 신호 사이의 변화하는 시간을 의미함. setup time은 1로 가는 시간, hold time은 0으로 가는 시간.

여담으로 intel의 cpu는 storage element(레지스터, 캐시)를 회로 중간에 심어 Critical Path Delay를 줄임. 다만 이러면 비용이 많이 들어감.



2.2.3. 실행시간 측정

Definition 10 *clock(cpu의 실행 시간)으로 프로그램의 실행 시간을 측정할 수 있음. 즉, 실행 시간을 아래의 식으로 구할 수 있음. 물론 Clock_cycle_time(주기) 대신 Clock_rate(진동수)를 사용할 수도 있음.*

```
Execution_time = Clock_cycles_for_program * Clock_cycle_time
```

하지만 아키텍처, 프로세서마다 clock이 다를 수 있기 때문에(ex. single cycle vs. multi cycle) 실행 시간을 이렇게 구하면 직관적이지 않음. 그래서 CPI를 활용하여 표기함.

2.2.4. CPI를 활용한 표기

Definition 11 *instruction 당 clock cycle을 CPI(Cycles per Instruction)라고 함. 즉, 실행 시간을 아래의 식으로 구할 수 있음. 물론 CPI의 역수인 IPC(Instructions per Cycle)를 사용할 수도 있음.*

```
Execution_time = #_of_Instructions * CPI * Clock_cycle_time
```

instruction마다의 clock cycle을 구하고 그 평균값을 CPI로 사용함. instruction마다의 clock cycle(각각의 CPI)을 고려하면 전체 cycle 수는 아래와 같으므로 평균 CPI를 구할 수 있음. 평균 CPI를 반영하면 실행 시간도 간단히 구할 수 있음.

```
Average_CPI = ((CPI_1 * #_of_In_1) + ... + (CPI_n * #_of_In_n))
                                                / #_of_Instructions
Execution_time = ((CPI_1 * #_of_In_1) + ... + (CPI_n * #_of_In_n))
                                                * Clock_cycle_time
```

이때 하나의 프로세서 내애서 명령어 별로 다른 cycle 수를 가지고 있을 수 있기 때문에, CPI 값을 instruction들의 평균으로 지정해야 함.

서로 다른 IPA의 비교를 위해 CPI를 사용할 수 있음. 물론 CPI가 언제나 정확한 지표로서 기능하지는 않음.

Instruction의 개수, CPI, clock cycle time을 줄여서 성능을 개선(사간 단축)할 수 있음. 디자이너는 여러 tradeoff를 잘 고려해서 instruction, cycle 수 등을 설계함.

ex. 프로그램에 사용된 instruction의 비율이 아래와 같을 때, 평균 CPI(그냥 CPI로 표기)와 실행 시간은 아래와 같음.

CPI = 0.43 * 1 + 0.21 * 2 + 0.12 * 2 + 0.24 * 2 = 1.57
Execution_time = #_of_Instructions * 1.57 * Clock_cycle_time

Instruction Class	Frequency	CPI_i
ALU operations	43%	1
Loads	21%	2
Stores	12%	2
Branches	24%	2

2.2.5. 계층별 성능 개선

clock의 관점에서, 아래와 같이 계층별로 영향을 미칠 수 있는 개선점이 다름.

이때 organization은 프로세서 구성을, technology는 하드웨어적인 기술을 말함.

코드를 개선하거나 좋은 컴파일러를 사용하면 instruction 수를 줄일 수 있음.

CPI와 Clock rate는 하드웨어적인 부분에서 주로 개선을 할 수 있음. 즉, 소프트웨어만 고려한 성능의 개선에는 한계가 존재함. (ex. OpenAI의 칩 개발 시작)

	Instruction count	CPI	Clock rate
Program	V		
Compiler	V		
ISA	V	V	
Organization		V	V
Technology			V

2.3. MIPS

clock이 시간을 기준으로 성능을 평가했다면, MIPS는 instruction 처리량을 기준으로 성능을 평가함.

2.3.1. MIPS

Definition 12 *MIPS(Million Instructions per Second)는 초당 instruction 처리량으로 성능을 나타내는 지표임. MIPS는 아래와 같이 구함.*

```
MIPS = Instruction_count / (Execution_time * 10^6)
```

이때 시간은 clock으로 구할 수 있음.

MIPS는 초당 실제 처리 instruction 개수가 아니라 10^6으로 나눈 개수임. 예를 들어 MIPS가 350이면 350 million(백만) 개의 instruction을 처리한 것. 당연히 데이터가 백만 단위로 주어지면 10^6으로 나눠줄 필요 없음.

clock을 짧을수록, MIPS는 클수록 빠른 것.

MIPS는 instruction set을 고려하지 않아 정확하지 않을 수 있음. CISC와 RISC를 생각하면 당연함.

2.3.2. MFLOPS

Definition 13 *MFLOPS(Million Floating-point Operations per Second)는 초당 부동소수점 연산(부동소수점 덧셈/곱셈/뺄셈 등) 처리량으로 성능을 나타내는 지표임.*

주로 부동소수점 곱셈을 사용함.

2.3.3. 연산 표기

ALU는 산술 연산을, load는 데이터를 가져오는 연산을, stroe는 저장 연산을, branch는 분기 연산을 의미함.

이때 ALU는 내부 회로만 사용하여 빠르지만, load, store는 메모리에 접근하기 때문에 느림. branch는 조건 분기 때문에 느림.

2.4. 벤치마크

2.4.1. 벤치마크

Definition 14 *성능을 시험하여 수치화한 것을 벤치마크(Benchmark)라고 함. 특정 instruction 집합이나 코드를 이용해 테스트하고, 여러 가지 지표를 종합하여 점수로 나타내는 것.*

시뮬레이션을 위해 사용되는 이런 instruction 집합이나 코드를 워크로드라고 함.

2.4.2. 벤치마크의 요건

벤치마크가 제대로 된 점수로서 기능하려면 아래의 요건들을 만족해야 함.

Relevant: 타겟 시스템 (e.g., 모바일, 서버)에서 실제로 수행될법한 워크로드를 사용해야 함.
Understandable: 결과를 쉽게 이해할 수 있어야 함.
Good Metric(s): 알아보기 쉽게 표기해야 함. 사용자들은 보통 점수를 선호함.
Scalable: 시스템 또는 어플리케이션이 달라져도 사용할 수 있어야 함.
Coverage: 타겟 시스템에서 중요한 것을 Oversimplify하지 않아야 함.
Acceptance: 판매자와 사용자 모두 납득할 수 있어야 함.

2.5. 성능 향상 반영 방법

2.5.1. Amdahl's Law

Definition 15 *실행시간의 향상은 1. 얼마나 향상되었는지 2. 향상된 부분이 전체에 얼마나 영향을 끼쳤는지에 의해 결정된다는 법칙을 Amdahl's Law(암달의 법칙)라고 함. 이를 반영한 식은 아래와 같음.*

```
New_time = Old_time * ((1 - Fraction) + (Fraction / Enhanced_degree))
```

Amdahl's law는 무어의 법칙과 더불어 가장 중요한 법칙 중 하나임.

즉, 어떤 부분이 향상되었으면, 해당 부분이 전체에서 차지하는 비율만큼 반영해주는 것. 예를 들어, 10%에 해당되는 연산의 성능이 2배 향상되었으면 개선된 전체 실행 시간은 New_time = Old_time * (0.9 * (0.1 / 2))로 구할 수 있음.

Part II

수의 처리

1. 정수의 처리

1.1. 정보의 저장

1.1.1. 정보의 저장

컴퓨터에서 정보는 여러 개의 비트와 자료형으로 구성됨. 이진 숫자로 정보를 표현하는데, 자료형이 그 표현과 해석의 기준이 되는 것.

1.1.2. 바이트 단위 인코딩

메모리는 바이트 단위로 주소가 부여되기 때문에 바이트 단위로 정보를 표기하는 것이 편리함. 이진수, 8진수, 16진수로 주로 표현하고, 10진수도 사용은 가능함. 각 진법은 환경별로 다르게 구분하지만, 대체로 0b, 0, 0x 등으로 구분함.

1바이트는 진법별로 아래와 같이 표기함.
Binary : 00000000 ~ 11111111
Octal : 000 ~ 888
Decimal : 0 ~ 255
Hexadecimal : 00 ~ FF

16진수를 주로 사용함.

1.2. unsigned

1.2.1. unsigned

unsigned 정수는 단순히 이진수로 바꾸어 져장함.

1.3. signed

1.3.1. 부호비트의 사용

signed 정수는 부호 비트(sign bit)를 사용해 부호를 표기함. 주로 최상위 비트(가장 왼쪽 비트)가 부호 비트로 사용됨. 양수는 부호 비트가 0, 음수는 부호 비트가 1임. unsigned와의 유사성을 지키기 위해 양수가 0, 음수가 1인 것.

부호 비트(최상위 비트)는 MSB(most significant bit)라고도 함. 가장 중요한 비트. 최하위 비트는 LSB(least significant bit)라고도 함.

양수는 unsigned와 같이 표기하면 되는데, 음수는 부호-크기 표현, 1의 보수, 2의 보수 등 방법이 여러 가지임. 물론 현대의 대부분의 컴퓨터에서는 2의 보수를 사용함.

1.3.2. 부호-크기 표현

Definition 16 *첫번째 비트로 부호를 표기하고, 나머지 비트로 수를 표기하는 방법을 부호-크기 표현(Sign-magnitude Representation)이라고 함.*

0을 표현하는 방법이 00...0과 10...0 두 가지이므로 불완전함.

부동소수점 연산에서는 이 방법을 사용하지만 정수에서는 사용하지 않음.

ex. 3은 0011, -3은 1011

1.3.3. 1의 보수

Definition 17 *비트를 전부 뒤집(NOT 연산)었을 때 양수와 음수가 대응되도록 표기하는 방법을 1의 보수 표현(One's Complement Representation)이라고 함.*

0을 표현하는 방법이 00...0과 11...1 두 가지이므로 불완전함.

당연하게도 1의 보수에서도 부호비트가 존재함.

여기서 보수는 뒤집어 표현하는 것을 말함.

ex. 3은 0011, -3은 1100.

1.3.4. 2의 보수

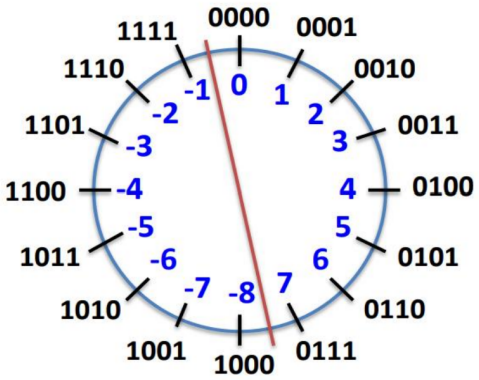
Definition 18 *비트를 전부 뒤집(NOT 연산)고 1을 더했을 때 양수와 음수가 대응되도록 표기하는 방법을 2의 보수 표현(Two's Complement Representation)이라고 함.*

0이 00...0으로 유일하게 표현됨.

당연하게도 2의 보수에서도 부호비트가 존재함.

비대칭적인 표현 범위를 가지기 때문에 0이 유일하게 표현됨.

현대의 대부분의 컴퓨터에서는 2의 보수를 사용함.



1.3.5. Sign Extension

sign 데이터를 더 큰 공간에 저장하면, 부호 비트가 복사되어 왼쪽에 추가됨. 이렇게 늘리면 2의 보수가 제대로 작동함.

ex. 0010은 0000 0010, 1010은 1111 1010.

1.4. 정수의 연산

C언어 기반으로 정리함.

1.4.1. 비트 연산

&, |, ∼, ∧로 모든 정수 자료형 데이터에 대한 비트 연산을 할 수 있음.

1.4.2. 논리 연산

&&, ||, !로 논리 연산을 할 수 있음.

이때 early termination이 적용되어, 판단이 끝나면 더 이상 판단을 하지 않음.

1.4.3. 시프트 연산

«, »로 시프트 연산을 할 수 있음.

오른쪽으로 이동하면 빈 공간에는 0이 채워짐. 왼쪽으로 이동하면 logical shift인 경우 0이, arithmetic shift인 경우 부호 비트가 채워짐.

0보다 작은 수 또는 데이터 길이보다 긴 수 만큼 시프트 연산을 하는 것은 정의되어 있지 않음.

곱셈 연산은 하드웨어적으로 부담이 심하기 때문에, 2의 거듭제곱을 곱하거나 나누는 것을 시프트연산으로 대체할 수 있음.

1.4.4. 덧셈/곱셈 연산

실제로 덧셈/곱셈은 이진수끼리 이뤄지는데, 그 방식은 10진수의 덧셈과 동일함. 연산 결과가 overflow된 경우 넘친 값은 무시하면 됨. 물론 그 결과가 주어진 비트로 표현할 수 없는 크기의 수라면 실제 값과는 다른 값으로 overflow됨.

오버플로우가 일어날 수 있음. 이때 signed냐 unsigned냐에 따라 오버플로우된 결과가 다르게 해석됨. 당연하게도 오버플로우는 곱셈에서 더 쉽게 일어남.

컴퓨터구조의 측면에서 보면 부호확장을 우선 하고, 그 이후에 연산을 수행함.

2. 실수의 처리

2.1. 실수 표현법

2.1.1. 실수 표현법

실수를 표현하는 방식으로는 고정소수점(Fixed-point)과 부동소수점(Floating-point)이 있음.

2.2. 고정소수점

2.2.1. 고정소수점

Definition 19 *고정소수점(Fixed-point) 방식은 소수점 위치를 고정하여 실수를 표현하는 방식임.*

정수부와 소수부의 비트 수를 미리 약속한 후 표현함. 이때 정수부는 n번째 비트의 경우 2의 n-1승에 해당되고, 소수부는 소수점으로부터 n번째 비트의 경우 2의 -n승에 해당됨. 각 자리의 거듭제곱들을 전부 더하면 해당 소수가 되는 것.

정수부와 소수부 비트 사이에 binary point(2진 소수점)가 들어감. 이때 binary point는 비트 표현에 포함되지 않으므로 해당 위치에 존재한다는 약속만이 존재함.

실수의 표현 시에는 부동소수점이 주로 쓰이지만, 고정소수점도 종종 쓰임.

예를 들어, 6.75는 0110110으로 표현됨. binary point를 표기하면 0110.110임.

2.2.2. signed 표현법

Definition 20 *고정소수점의 signed 표현법으로는 부호비트를 사용하는 것(sign-magnitude)과 2의 보수로 표현하는 것이 있음.*

부호비트와 2의 보수는 정수에서의 방식과 동일함.

2.3. 부동소수점

2.3.1. 부동소수점

Definition 21 *부동소수점(Floating-point) 방식은 소수점 위치를 가장 큰 자릿수의 숫자 바로 오른쪽으로 고정하여 표현하는 방식임.*

소수점 위치를 가장 큰 자릿수의 바로 오른쪽으로 지정하기 위해 특정 수의 거듭제곱을 곱한 형태로 실수를 나타냄. 즉, $\pm M \times B^E$ 꼴로 표현됨. 이때의 M을 mantissa(가수부), B를 base(밑), e를 exponent(지수부)라고 함. 이때 가수부는 수의 정밀도와, 지수부는 수의 범위와 관련이 있음.

부동소수점 표현 방식은 사용하는 비트 수에 따라 단정밀도와 배정밀도로 나눌 수 있음.

1. 32비트 크기의 부동소수점 표현 방식을 단정밀도(single precision)라고 함. 단정밀도는 1비트의 부호 비트, 8비트의 exponent, 23비트의 mentissa를 가지고, bias는 10진수로 127임. c에서의 float이 단정밀도 방식을 사용함.

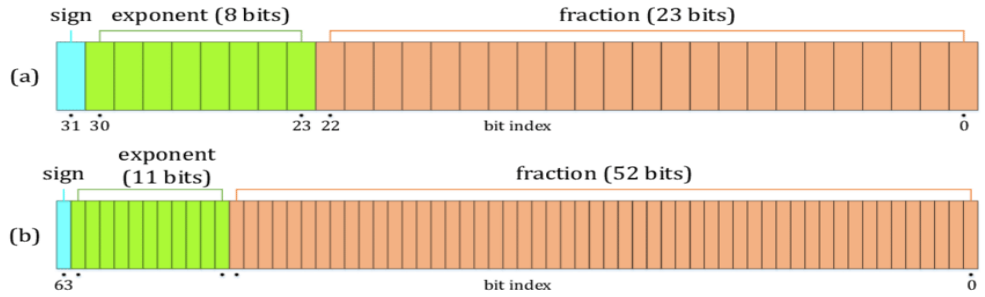
2. 64비트 크기의 부동소수점 표현 방식을 배정밀도(double precision)라고 함. 배정밀도는 1비트의 부호 비트, 11비트의 exponent, 52비트의 mentissa를 가지고, bias는 10진수로 1023임. c에서의 double이 단정밀도 방식을 사용함.

부동소수점의 방식은 여러 가지가 있지만, 여기서는 IEEE 754 단정밀도 부동소수점 표현 방식을 정리함.

예를 들어, $273 = 2.73 \times 10^2$등으로 나타내는 것임.

부호 비트 1비트, exponent 8비트, fraction 7비트를 쓰는 google의 bfloat16이 존재하는 것처럼, 다양한 부동소수점 표현 방식이 존재함. 이 경우 IEEE 754 단정밀도와 exponent의 크기가 같으므로 정밀도는 낮지만

더 적은 비트 수로 동일한 범위의 수를 표현할 수 있음.



2.3.2. IEEE 754 단정밀도 부동소수점 표현 방식

Definition 22 *IEEE 754 단정밀도 부동소수점 표현 방식은 단정밀도 공간에 1비트의 부호 비트, 8비트의 exponent값, 23비트의 mentissa값을 저장하지만, 몇 가지 기법들이 추가로 사용됨.*

IEEE 754 단정밀도 부동소수점은 아래의 방식으로 실수를 표현함. base는 2를 사용함.

1. 실수를 이진수으로 표기
우선 고정소수점 표기법으로 실수를 이진수로 표기하고, 2의 거듭제곱을 곱한 형태(shift)로 나타내 binary point를 가장 높은 자릿수의 바로 오른쪽으로 이동시킴. 이후 부호를 부호 비트에, 2의 지수부를 이진수로 나타내 exponent에, mantissa에 최상위 비트부터 가수부를 채워 넣음.

예를 들어, 228(1110 0100)을 1.11001×2^7*로 나타내고, sign bit에 0, exponent에 111, mantissa에 111001000...0을 넣음.*

2. mantissa의 최상위 비트 생략
binary point를 가장 높은 자릿수의 바로 오른쪽으로 이동시키므로, mantissa에 저장되는 최상위 비트는 항상 1임. 이때의 1을 leading 1이라고 함. 1을 작성하지 않고 1이 항상 존재하는 것으로 취급하여 1비트를 아낄 수 있음. 즉, 소수점 아래 부분만을 mantissa에 작성함. 이렇게 leading 1을 제외하면 이 부분을 fraction이라고 함. 이때 fraction은 왼쪽(높은 주소 방향)부터 넣음.

예를 들어, 111001000...0을 넣는 대신 110010000...0을 넣음.

3. Biased Exponent 사용
빠르고 간단한 비교와 연산을 위해 bias라는 개념을 사용함. bias는 양수 값과 음수 값을 모두 표현하기 위한 약속된 값임. exponent 값에 bias를 더한 값을 exponent로 작성하는데, 이를 biased exponent라고 함. 8비트를 위한 bias는 0111 1111임. exponent의 비트가 n개이면, n-1개의 1로 채워진 비트와 최상의 비트 0의 조합(총 n개)으로 bias가 결정됨. 당연하게도 biased exponent로 저장된 경우 원래의 값을 얻으려면 bias만큼을 빼 줘야 함.

예를 들어, exponent가 111인 경우 0111 1111을 더한 값인 1000 0110가 biased exponent로 exponent 자리에 넣음.

정리하면, IEEE 754 단정밀도 부동소수점 표현 방식은 아래의 과정을 거쳐 실수를 표현함.

1. 고정소수점 방식으로 실수를 이진수로 표현함.
2. 부호에 따라 부호 비트를 넣음.
3. bias exponent를 넣음.
4. fraction을 넣음.

각 기법의 적용 유무에 따라 다른 부동소수점의 표현 방식으로 생각할 수도 있음.

본 필기는 IEEE 754 단정밀도 부동소수점 표현 방식을 기준으로 작성되었음.

2.3.3. 부동소수점 rounding

Definition 23 *값을 특정 값으로 근사시키는 것을 rounding이라고 함.*

단정밀도와 배정밀도로는 모든 실수를 표현하는 것이 불가능하기 때문에, 숫자가 너무 크면 overflow

가, 숫자가 너무 작으면 underflow가 일어남. 이에 따라 내부적으로 적절한 rounding이 수행됨.

단순한 올림/버림, 0에 가까운 쪽으로 올림/버림, 가장 가까운 수로 올림/버림 하는 방법 등이 존재함.

부동소수점에서 수의 정밀도를 나타내는 것은 가수부임. 즉, fraction에 해당하는 비트 수로 나타낼 수 없는 실수에 대해서 rounding이 필요한 것.

예를 들어, 1.100101을 fractional bit를 3개만 쓰도록 rounding한다고 하면, 올림 시에는 1.101, 내림 시에는 1.100, 0에 가까운 쪽으로는 1.100, 가장 가까운 수로는 1.101로 rounding됨.

2.3.4. 정규화된 수와 비정규화된 수

Definition 24 *(biased) exponent가 000...0, 111...1이 아닌 일반적인 경우의 부동소수점 표현은 정규화되어(normalized) 있음. 지금까지 살펴본 표현 방식은 정규화된 수(normalized value)를 나타내는 방식임.*

(biased) exponent가 000...0, 111...1인 경우는 예외적인 상황으로, 비정규화된 수(Subnormalized value)를 표현하도록 약속되어 있음.

exponent가 000...0인 경우에는 fraction이 000...0인지, 000...0이 아닌지에 따라 다르게 취급됨. exponent가 111...1인 경우는 특수한 경우에 대한 표현으로 사용되는데, 이는 뒤에 따로 정리함.

1. exponent가 000...0이고, fraction이 000...0인 경우
0.0으로 취급함. 이때 부호 비트에 따라 +0과 -0이 모두 가능함.

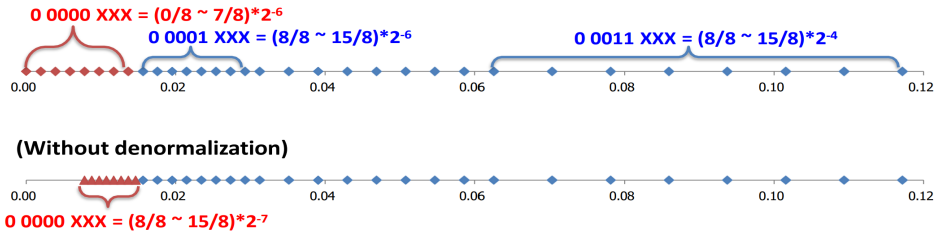
2. exponent가 000...0이고, fraction이 000...0이 아닌 경우
exponent와 mantissa를 특수하게 설정함.
-> 저장된 값으로 실제 값을 나타낼 때 exponent 값은 biased exponent - bias가 아니라 1 - bias로 계산함. 즉, 저장된 exponent 값에 +1한 값을 적용함. 또한 mantissa의 생략된 첫번째 비트 수를 leading 1 대신 0으로 계산함. 즉, 저장된 mantissa에 -1한 값을 적용함.

표현 가능한 실수 값을 차례대로 나열해 보면, 각 exponent에 대해 mantissa(fraction)은 항상 균일하게 커지고 작아지는데에 반해, exponent 값은 2의 거듭제곱으로 적용되므로 그 값이 점점 급격하게 커지고 작아짐. 수직선 상에 표현이 가능한 수들을 찍어 보면 0에 가까워질수록 값이 급격하게 작아져 일부 수들을 표현할 수 없게 됨. 이에 따라 mantissa의 생략된 첫 번째 비트를 0으로 하고 적용되는 exponent 값을 1 크게 함으로써 0과 가까운 각 실수들의 간격을 고르게 할 수 있음. 이를 점진적 underflow(gradual underflow)라고 함.

또한 이런 기법을 적용하면 더 작은 소수까지 표현할 수 있음.

순차적인 값을 비교해 보면 점진적 underflow에 대한 이해가 쉬움. 필요하다면 Number 2 ppt 맨 마지막을 참고하자. 7비트 크기의 부동소수점 표현을 사용한 예시가 있음.

실제로 bias exp가 1인 구간의 mantissa를 $1 + a_1, \cdots, 1 + a_n$이라고 하고, bias exp가 0인 구간의 mantissa를 $a_1, \cdots, a_n$이라고 하면 각 구간에서 원소별 간격을 계산할 수 있음. graducal underflow를 적용하면 두 구간 모두 간격이 $(a_{n+1} - a_n) \times 10^{-126}$으로 같지만, 적용하지 않은 경우 bias exp가 0인 구간은 간격이 $(a_{n+1} - a_n) \times 10^{-127}$이 되어 더 좁음.



2.3.5. 특수한 경우에 대한 표현

Definition 25 *아래와 같이 exponent가 111...1인 경우는 특수한 경우에 대한 표현으로 사용됨.*

∞ *: 부호 비트가 0, exponent가 111...1, fraction이 000...0.*

$-\infty$: 부호 비트가 1, exponent가 111...1, fraction이 000...0.
NaN : 부호 비트가 0 또는 1, exponent가 111...1, fraction이 0이 아닌 값.

NaN(Not a Number)은 값을 실수로 정의할 수 없음을 나타냄. (ex. sqrt(-1), log(-5))

2.3.6. 부동소수점의 덧셈

부동소수점의 덧셈은 아래의 과정을 거쳐 수행됨.

1. (biased) exponent와 fraction을 추출함.
2. fraction에 leading 1을 추가하여 mantissa로 복원함.
3. exponent가 더 작은 쪽의 mantissa를 시프트하여 두 실수의 exponent를 맞춤.
4. mantissa끼리 덧셈을 수행함.
5. 필요하다면 결과값에서 exponent와 mantissa를 조정해 binary point를 앞으로 당김.
6. exponent/fraction의 비트가 부족할 경우 결과를 rounding함.
7. exponent와 fraction을 부동소수점 format에 삽입함.

복잡한 과정을 거쳐야 하므로 프로세서의 성능 측정 시에 부동소수점 연산을 지표로 사용하기도 하는 것. 연산의 수행을 위해 몇 cycle이 필요함.

2.3.7. 부동소수점의 곱셈

부동소수점의 곱셈은 아래의 과정을 거쳐 수행됨.

1. 두 exponent를 더함.
2. 두 mantissa를 곱함.
3. 필요하다면 결과값에서 exponent와 mantissa를 조정해 binary point를 앞으로 당김.
4. exponent/fraction의 비트가 부족할 경우 결과를 rounding함.
5. exponent와 fraction을 부동소수점 format에 삽입함.

덧셈과 곱셈은 비슷한 정도의 시간 복잡도를 가지지만, 곱셈에서는 곱셈 연산이 중간에 수행됨.

2.3.8. c에서의 부동소수점

c에서는 float으로 단정밀도 표현을, double로 배정밀도 표현을 지원함.

double, float을 int로 변환 시에는 fraction 부분을 버리는(0쪽으로 rounding하는 것) 과정을 거치고, 반대 변환의 경우에는 주어진 비트 내에서 적절한 rounding을 수행함.

소수의 연산은 표현의 한계 때문에 그 결과가 정확하지 않음. 특히, 연산 시에는 큰 소수끼리 먼저 계산하는 것이 좋음.

참고로, c에서의 자료형 크기는 32비트 os냐, 64비트 os냐에 따라 달라짐. 다른 자료형들은 크기가 동일한데, long과 포인터 자료형의 크기가 32비트에서는 4바이트이고 64비트에서는 8바이트임.

Part III

ISA

1. ISA

1.1. cpu와 어셈블리어

1.1.1. 어셈블리어

Definition 26 *어셈블리어는 대체로 하나의 문장이 하나의 기계어와 대응되는 Low_level 언어임.*

High_level 프로그램과 기계어 사이의 Abstraction layer로서 기능함.

기계어는 cpu 종속적이기 때문에, 어셈블리어도 cpu 종속적임.

1.1.2. cpu의 구조

Definition 27 *cpu(Central Processing Unit)은 아래와 같이 크게 4가지 구성요소를 가지고 있음.*

1. PC(Program Counter) : 현재 실행 중인 명령어의 주소를 가리키는 레지스터.(제어)
2. 레지스터 파일 : 레지스터.
3, ALU (Arithmetic/Logic Unit) : 산술/논리 연산을 수행하는 회로.
4. Control logic : cpu 동작을 제어하는 부분. fetch(명령어 가져옴), decode(명령어 해석), execute(명령어 실행) 등을 수행함.

현존하는 대부분의 cpu들은 이 구조를 따름. cpu는 명령어를 순차적으로 실행하고, 내부적으로 다양한 기법들을 사용함.

cpu의 작동은 아래 과정의 반복임.
1. PC가 가리키는 명령어를 fetch
2. decode
3. execute
4. update PC

데이터 이동의 관점에서는 아래 과정의 반복으로도 볼 수 있음.
1. memory -> register
2. execute
3. register -> memory

1.2. ISA

1.2.1. ISA

Definition 28 *ISA(명령어 구조 체계, Instruction Set Architecture)는 CPU가 어떻게 작동하는지에 대한 규격임.*

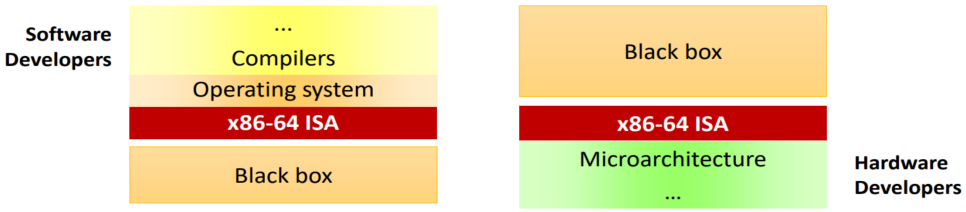
명령어(instruction set), 레지스터, 자료형, 주소 지정 방식, 바이트 오더링, 인코딩 방식 등을 포함함.

isa는 소프트웨어(os, compiler 등)와 하드웨어 사이의 추상화, 인터페이스로서 기능함. 소프트웨어에서는 하드웨어를 어떻게 조작할지를 생각하고, 하드웨어에서는 소프트웨어의 요구사항에 맞도록 구현하는 것을 생각할 수 있음.

cpu마다 각각의 isa와 그에 따른 명령어(instruction set)를 사용함.

isa는 명령어와 그 구조에 대한 설계임. 명령어에 따라 적절한 레지스터가 존재해야 하기 때문에 레지스터에 대한 내용이 포함되어 있고, 명령어가 메모리를 조작해야 하기 때문에 자료형, 메모리 주소 지정 방식, 바이트 오더링, 인코딩 방식 등에 대한 내용이 포함되어 있음.

소프트웨어의 요구사항을 만족하면서 효율적인 하드웨어를 구현하는 isa가 이상적임.



1.2.2. n비트 ISA

Definition 29 *n비트 isa라고 할 때 n비트는 명령어 하나를 표현하는 데에 필요한 비트 수임.*

128비트 isa도 존재는 하지만, 현재는 거의 사용되지 않음.

1.2.3. 레지스터

Definition 30 *마이크로프로세서가 사용하는 가장 작고 빠른 메모리를 레지스터(register file)라고 함.*

n비트 프로세서인 경우 각 레지스터는 n비트의 크기를 가짐.

레지스터의 각 비트에 대해 이야기할 때는 c의 배열과 같이 0번째~63번째 등으로 표현함.

메모리로의 접근과 레지스터로의 접근은 그 성능 차이가 100배 정도임.

보조기억장치에 있는 코드 등을 메모리에 올리고(프로세스), 메모리의 값을 레지스터로 가져와 산술/논리 연산을 수행함.

이때 메모리는 바이트 단위 주소를 가지고, 코드와 데이터 등을 저장함. 또한 함수 등에 대한 스택이 존재함.

컴파일러는 레지스터 사용을 최적화하여 가능한 한 많은 레지스터를 효율적으로 운용해야 함.

본 수업에서는 기본적으로 64비트 프로세서에 대해 다루고, 그에 따라 레지스터 크기를 64비트로 생각함.

1.2.4. isa 디자인 원칙

isa의 연산 디자인에는 몇 가지 원칙이 있음.

1. Simplicity favors regularity(단순함이 규칙성에 유리함)
단순하게 디자인하면 구현을 간단하게 하고, 적은 비용으로 높은 성능을 낼 수 있음.

2. Smaller is faster(작을수록 빠름)
크기가 작아질수록 속도가 빨라짐. 보조기억장치보다 메모리가, 메모리보다 캐시 메모리가, 캐시 메모리보다 레지스터가 빠름. 레지스터의 성능 자체도 더 좋지만, 연산 하나당 가져야 하는 비트 수가 늘어나면 cpu가 처리해야 하는 데이터의 양도 커지므로 속도가 느려짐.

3. Good design demands good compromises(좋은 디자인은 타협을 요구함)
최적화된 설계를 위해 일부 불편함은 감수함. 예를 들어, Instruction format에서 구간을 이상하게 쪼개는 것은 디코딩 시의 성능 향상을 위한 것임.

4. Make the common case fast (Amdahl's Law)

2. RISC-V

2.1. RISC-V

2.1.1. RISC-V

Definition 31 *RISC-V는 UC Berkeley에서 개발한 5번째 risc 아키텍처로, RISC-V 아키텍처의 isa는 학계/산업계에서 무료로 이용 가능한 완전 오픈 isa임.*

명령어의 길이에 따라 RV32I(RISC-V integer), RV64I, RV128I로 기본 integer(정수용) isa가 3개 존재함. RV32E 같이 레지스터의 개수를 16개로 줄인 RV32I의 임베디드 전용 버전도 존재함.

RV32I는 아래와 같이 여러 확장들을 제공함.
M : integer Multiply/Divide 제공.
A : atomic instructions 제공.
F : single-precision floating point(단정밀도 부동소수점) 제공.
D : double-precision floating point(배정밀도 부동소수점) 제공.
G : general purpose. IMAFD가 적용됨.

Q : quad-precision floating point 제공.
C : compressed instructions. 명령어의 길이를 16비트로 줄인 버전. 이는 32비트 명령어와 호환됨.

M extension에는 mul(곱셈) 연산 등이 존재함.

RV32I에는 47개의 명령어들이 존재하는데, x86은 1503개, arm은 500정도인 것을 생각하면 47개는 굉장히 적은 것임. arm은 risc이지만 RISC-V에 비하면 risc에 덜 가까움.

compressed instruction들을 제공하는 C extension과의 호환성을 위해 RV32I는 주소를 2 byte 단위로 alignment함.

현재 risc 기반 isa는 임베디드에서 자주 사용됨.

본 수업에서는 RV32I를 다룸.

2.1.2. RISC-V의 레지스터

RV32I는 아래와 같이 32개의 레지스터들과 pc를 가짐.

#	Name	Usage
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporaries (Caller-save registers)
x6	t1	
x7	t2	
x8	s0/fp	Saved register / Frame pointer
x9	s1	Saved register
x10	a0	Function arguments / Return values
x11	a1	
x12	a2	Function arguments
x13	a3	
x14	a4	
x15	a5	

#	Name	Usage
x16	a6	Function arguments
x17	a7	
x18	s2	Saved registers (Callee-save registers)
x19	s3	
x20	s4	
x21	s5	
x22	s6	
x23	s7	
x24	s8	
x25	s9	
x26	s10	
x27	s11	
x28	t3	Temporaries (Caller-save registers)
x29	t4	
x30	t5	
x31	t6	
	pc	Program counter

zero는 항상 0(Hard-wired zero)을 저장하고 있는 레지스터로, 값을 수정할 수 없음. 0을 사용 시에 사용함.

ra에는 호출 종료 후 제어가 복귀할 주소를 저장함.

sp에는 현재 스택의 어떤 위치를 사용 중인지 저장함.

s0~s11은 호출된 함수들이 사용하는 레지스터임.

t0~t6는 임시 레지스터로, 일시적인 사용 등을 위한 중간 공간으로 사용함.

a0~a7은 함수의 인자로 작성한 값을 저장하는 레지스터임. 인자로 받는 값이 9개 이상이면 값을 메모리에 저장하고 꺼내야 하기 때문에 성능이 급격히 떨어짐. a0과 a1에는 반환값을 저장하는데, 레지스터가 2개인 이유는 각 레지스터가 32비트 크기이기 때문에 64비트 크기의 데이터를 반환할 때 a0와 a1 모두를 사용해야 하기 때문임.

gp, tp는 이런 것이 있다 정도만 알자.

3. RISC-V 연산

레지스터의 값을 이용한 산술/논리 연산, 데이터 이동 연산, 분기 및 호출/반환 연산이 존재함.

연산 시에 레지스터는 최소로 사용하도록 최적화하되는 것으로 보임.

3.1. 연산과 데이터

3.1.1. 연산의 내부 데이터 구조

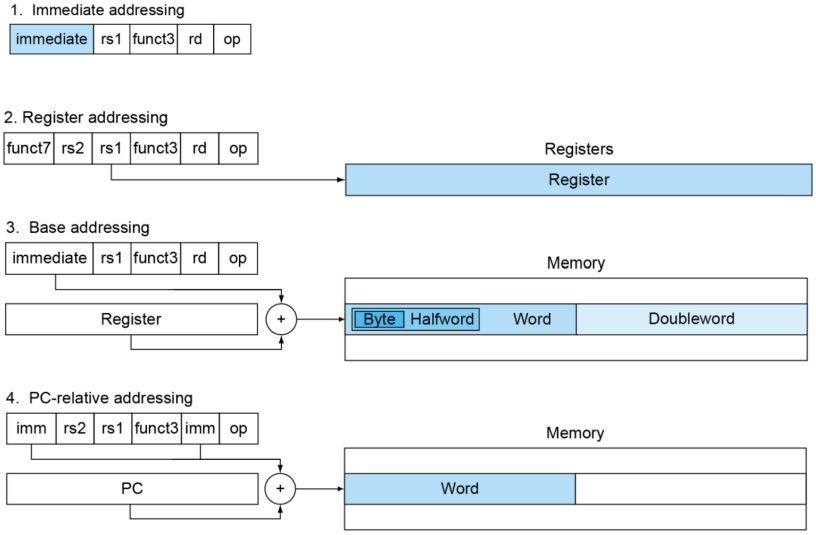
RV32I의 연산은 32비트로 표현되고, 각 비트들의 구간은 나름의 역할을 가짐. 이때 각 구간이 가지는 값의 형태는 4가지로 분류할 수 있음.

1. immediate value(상수)
: 자주 사용되는 값을 12비트로 직접 표현해 사용하는 부분. immediate value를 사용하면 상수를 따로 레지스터를 저장하여 사용하지 않아도 됨.
immediate는 signed임.

2. register
: 레지스터 주소.
각 레지스터는 명령어 수행 시에 임시 변수(automatic variable)로 사용되기도 함.

3. memory address
: 레지스터의 값과 immediate 값을 더한 값으로 메모리에 접근함.

4. pc address
: pc에 immediate를 더한 값으로 다음 연산에 접근/분기 등을 함.



3.1.2. 데이터 표현 단위

비트는 0과 1을 값으로 가지고, 이런 비트들의 집합으로 byte(8bits), half-word(16bits), word(32bits), double word(64bits)를 만듦.

32비트 크기의 데이터는 워드, 64비트 크기의 데이터는 double word라고 함.

기본적으로 integer(int)는 word의 크기(4바이트)를 가짐.

3.1.3. 바이트 오더링

Definition 32 *바이트 오더링에는 빅 엔디안과 리틀 엔디안이 존재함. 말 그대로 빅 엔디안은 높은 자릿수의 값이 끝(낮은 주소)에, 리틀 엔디안은 낮은 자릿수의 값이 끝에 들어가는 것을 말함.*

대부분의 아키텍처가 리틀 엔디안 방식을 따르고, RISC-V에서도 기본적으로는 리틀 엔디안 방식을 지원함.

서로 다른 바이트 오더링을 따르는 시스템 사이의 데이터 이동 시에는 문제가 발생할 수 있음.

Sun, PowerPC Mac, 네트워크 등에서는 빅 엔디안을, intel x86, ARM(Andorid, ios), RISC-V에서는 리틀 엔디안을 따름.

3.1.4. alignment

Definition 33 *메모리에 저장되는 데이터가 word 등을 기준으로 정렬되도록 하는 것을 alignment라고 함.*

alignment를 하는 것은 메모리를 더 소비하지만 성능적 측면에서는 더 뛰어남. unaligned를 사용하면 해당 데이터를 처리하는 데에 시간이 더 오래 걸리고, atomicity(원자성)가 부족해 미묘한 버그들이 발생할 수 있음.

RISC-V는 alignment를 강제하지는 않지만 성능적 문제로 인해 지키도록 권고함.

alignment를 강제하는 아키텍처에서는 반드시 지켜야 하지만, 초기의 아키텍처들에서 강제되던 alignment는 개발자에게 불편하다는 이유로 사라졌음. 현재는 RISC 등에서 성능을 위해 이를 일부 채택했음.

RISC-V는 alignment를 강제하지는 않지만, alignment만 지원하도록 설계되어 있고 unaligned의 처리에 대해서는 os의 도움을 추가로 받아야 함.

어떤 한 작업에 대한 연산이 여러 개로 쪼개졌을 때 모든 연산이 전부 통일성 있게 수행되야 하는데, 이를 atomic하게 연산이 수행되야 한다고 함. alignment가 맞지 않아 연산이 여러 개로 쪼개진 경우 모든 연산들 atomic하게 수행하기 힘들기 때문에 버그들이 발생하는 것임.

3.2. 산술/논리 연산

3.2.1. 산술 연산

Definition 34 *산술(arithmetic) 연산으로는 아래와 같은 것들이 있음.*

add : 덧셈.
sub : 뺄셈.
slt : set less than.(<) 참이면 1, 거짓이면 0을 대입.
sltu : 비교하는 피연산자들을 unsigned로 취급해 slt를 연산함.

RV32I의 모든 산술 연산은 아래와 같이 3개의 피연산자를 가짐. 뒤의 두 피연산자를 연산하여 맨 앞 피연산자에 대입하는 것. 표준 문서 등에 작성할 때 맨 앞 피연산자는 rd(register destination), 나머지 피연산자는 rs(register source)로 표기하기도 함.

피연산자에 immediate value(상수)가 존재하는 연산을 immediate operands라고 함. immediate값은 주로 rs2자리에 작성함. immediate opperands는 addi와 subi와 같이 뒤에 i가 붙은 연산을 사용함. 참고로, sltu의 immediate operands는 sltiu로 작성함. u가 더 늦게 붙음.

```
add rd, rs1, rs2                // rd = rs1 + rs2
addi rd, rs1, imm12             // rd = rs1 + imm12
slt rd, r1, r2                  // r1 < r2으로 rd에 1 또는 0 대입
```

3.2.2. upper immediate operation

Definition 35 *upper immediate operation은 12비트보다 더 큰 범위의 immediate를 사용하기 위한 산술 연산으로, 기존 immediate에 해당하는 12비트의 상위 위치에 20비트 immediate를 추가로 지정함.*

연산으로 지정한 20비트 immediate는 레지스터의 12번째~31번째 비트에 들어가고, 기존 immediate로 값을 넣을 수 있는 부분인 0번째~11번째 비트는 0으로 채워둠. 또한 64비트 레지스터 기준 32번째~63번째 비트는 sign extension하여 31번째 비트 값으로 채움.

upper immediate operation으론 아래와 같은 것들이 있음.

lui : load upper immediate.
auipc : add upper immediate to pc.

아래와 같이 작성함.

```
lui rd, imm20
```

```
auipc rd, imm20
```

upper immediate opperand는 12번째 비트부터 20비트를 채우기 때문에, 아래와 같이 실제로는 addi 등과 함께 사용함. 기존의 연산에서 immediate값이 12비트를 넘어가면 넘어간 만큼을 upper immediate opperand로 채워주는 것.

```
lui x19, 0x003D0
addi x19, x19, 0x500    // 결과적으로 0x003D0500이 x19 레지스터에 들어감.
```

lui x19, 0x003D0	0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000
addi x19,x19,0x500	0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

3.2.3. 논리 연산

Definition 36 *논리(logical) 연산으로는 아래와 같은 것들이 있음. 전부 비트 단위 연산이고, 동일한 형태로 작성함.*

sll : shift left logical. 빈 공간은 0을 채움.
sra : shift right arithmetic. 빈 공간은 MSB로 채움.
srl : shift right logical. 빈 공간은 0을 채움.
and : and bit 연산.
or : or bit 연산.
xor : xor bit 연산.

RV32I의 논리 연산은 아래와 같이 3개의 피연산자를 가짐. 뒤의 두 피연산자를 연산하여 맨 앞 피연산자에 대입하는 것.

논리 연산에도 immediate operands가 존재함. 연산 뒤에 i만 붙이면 됨.

```
sll rd, rs1, rs2                // rd = rs1 + rs2
slli rd, rs1, imm12             // rd = rs1 + imm12
```

RISC-V에는 곱셈/나눗셈 연산이 따로 존재하지 않아 시프트/덧셈/뺄셈 연산 등으로 이를 수행함.

and, or, xor 사용 예시로는, and 연산으로는 마스킹하여 특정 비트를 추출/삽입하는 데 사용할 수 있고, or 연산으로는 두 데이터 사이의 포함관계를 판단하는 데에 사용할 수 있고, xor 연산으로 두 데이터가 동일한지를 판단할 수 있음.

참고로 c에서 »는, unsinged 정수형에 대해서는 logical right shift 연산을 수행하고, signed 정수형에 대해서는 arithmetic right shift 연산을 수행함. 부호 비트가 존재하면 반영해 주는 것.

시프트 연산으로 곱셈을 수행하는 예시로, x5에 48을 곱하는 경우 아래와 같이 연산을 수행함.

```
slli x6, x5, 1      // 곱하기 2
add x6, x6, x5      // 원래의 수 더하기
slli x5, x6, 4      // 곱하기 16
```

3.2.4. 포인터 vs 배열

c문법 기준으로, 코드 작성 시에는 배열이 포인터보다 편리하지만 실제 수행되는 작업량은 배열이 더 많아 비효율적임.

배열의 경우 요소 크기만큼의 곱셈(시프트)을 해 주고, 배열의 시작 주소에 해당 값만큼 더해 각 요소에 접근함. 반면에 포인터는 단순히 지정한 주소로 이동함.

컴파일 시에 -O1 옵션을 지정하면 배열을 사용해도 포인터와 동일한 연산이 수행됨. 임베디드 시스템 등을 다룰 때는 이런 옵션으로 인해 잘못된 동작이 발생할 수 있긴 함.

배열 인덱스 지정 시에 곱셈 연산 대신 더 효율적인 시프트 연산을 주로 사용함.

3.3. 데이터 이동 연산

load와 store는 사용법이 거의 동일함.

3.3.1. load 연산

Definition 37 *메모리의 데이터를 레지스터로 이동하는 연산으로는 아래와 같은 것들이 있음. 전부 동일한 형태로 작성함.*

ld : load double word.(64비트)
lw : load word.(32비트)
lh : load half word.(16비트)
lb : load byte.(8비트)

ldu, lwu, lhu, lbu와 같이 연산 뒤에 u를 붙이면 unsigned에 대한 연산이 수행됨. 32비트 레지스터에 값을 가져올 때 데이터가 signed인지 unsigned인지에 따라 남은 공간에 sign extension을 수행할 것인지를 정하는 것.

load 연산의 피연산자는 destination register, base register(address), 그리고 immediate(상수) offset가 존재함. base register부터 offset만큼의 위치에서 연산 만큼의(word 등) 데이터를 destination register로 가져오는 것.

```
lw dst, off(base)
ld rd, imm12(rs1)
```

배열, 구조체, 동적 메모리 등은 그 값을 연산 전에 메모리에서 레지스터로 load해야 함.

3.3.2. store 연산

Definition 38 *레지스터의 데이터를 메모리로 이동하는 연산으로는 아래와 같은 것들이 있음. 전부 동일한 형태로 작성함.*

sd : load double word.(64비트)
sw : load word.(32비트)
sh : load half word.(16비트)
sb : load byte.(8비트)

store 연산의 피연산자는 destination register, base register(address), 그리고 immediate(상수) offset가 존재함. destination register의 데이터를 base register부터 offset만큼의 위치에 저장하는 것.

```
sw dst, off(base)
sd rd, imm12(rs1)
```

store는 주어진 레지스터 값을 메모리에 그대로 옮겨적기 때문에 unsigned 연산이 존재하지 않음.

3.4. 분기 연산

3.4.1. 조건부 연산

Definition 39 *조건이 참이면 분기하는 조건부 연산으로는 아래와 같은 것들이 있음. 전부 b(branch)로 시작하고, 동일한 형태로 작성함.*

beq : branch if equal.(==)
bne : branch if not equal.(!=)
blt : branch if less than.(<)

bgt : branch if greater than.(>)
ble : branch if less than or equal.(<=)
bge : branch if greater than or equal.(>=)

조건부 연산은 앞에 작성하는 두 개의 피연산자를 비교하여 참이면 맨 뒤의 label로 분기함. 이때 label은 이동할 지점을 나타내고, 기계어로 번역 시 주소로 변환됨. label에는 12비트 immediate를 작성할 수 있는데, 이 경우 immediate 만큼이 pc에 더해짐.

bequ, bneu, bltu, bgeu와 같이 연산 뒤에 u를 붙이면 피연산자들을 unsigned로 취급해서 연산함.

```
beq rs1, rs2, imm12     // 참이면 pc += SignExt(imm12 << 1)
bltu rs1, rs2, imm12
```

u를 붙일 경우, signed로 보냐 unsigned로 보냐에 따라 엄청 큰 수가 되거나 음수가 되기 때문에 유의해야 함.

3.4.2. 무조건 분기 연산

Definition 40 *무조건 분기하는 연산으로는 아래와 같은 것들이 있음. 전부 j(jump)로 시작함. 이 연산들은 프로시저 호출 시에 주로 사용함.*

jal : jump and link. 지정한 immediate 값으로 분기.
jalr : jump and link register. 특정 레지스터의 주소로부터 immediate만큼 떨어진 위치로 분기. 이런 분기를 computed jump라고 함.

jal과 jalr은 아래와 같은 형태로 작성함. jal은 rd에 호출 이후 실행할 다음 명령어를 작성하고(pc + 4 등) label(pc + imm20)로 분기함. jalr은 rd에 호출 이후 실행할 다음 명령어를 작성하고(pc + 4, x0 등) rs1으로부터 imm12만큼 이동한 위치로 분기함.

```
jal rd, imm20
jalr rd, imm12(rs1)
```

함수 호출 시에는 jal을, 반환 시에는 jalr를 사용함. jal로는 x1(ra)에 다음 명령어 주소를 작성하고 타겟 주소(label 등)로 분기함. jalr로는 0x(hard-wired zero)에 주소를 작성하고(x0은 수정될 수 없으므로 무시됨.) x1(ra)로부터 0만큼 떨어진 위치로 분기함. 즉, 아래와 같이 작성함.

```
jal ra, func        // func 호출
jalr x0, 0(ra)      // 반환
```

lui와 jalr를 함께 사용하면 총 32비트로 떨어진 위치를 지정할 수 있으므로, jal의 20비트로 표현이 불가능한 부분도 표현이 가능함. 아래와 같이 lui로 레지스터에 20비트 upper immediate를 넣고, 그 값을 jalr로 지정한 12비트 immediate를 더해 해당 위치로 이동하는 것.

```
lui x19, 0xABCDE    // 0xABCDE000이 저장됨
jalr 0x, 0xEFG      // 0xABCDEFG로 분기됨
```

3.4.3. 분기 주소 지정 방식

분기하여 이동할 위치의 주소 지정에는 3가지 특징이 있음. 이때 분기하여 이동하는 지점은 메모리 내의 특정 명령어임.

1. 분기할 위치에 대한 주소는 짝수임.
즉, 2바이트로 alignment가 됨.
RISC-V의 extension 중 c(compressed)에서는 명령어가 16비트 길이로 인코딩되기 때문에, 호환성을 위해 2바이트 단위로 alignment하는 것.

2. 조건부 분기문 주소 지정 방식
branch 시에는 pc에 12비트 immediate만큼을 더한 주소를 사용함.
if/for/while 등의 branch문에서는 대체로 분기할 주소가 분기 명령어 근처에 존재하기 때문.

3. 무조건 분기문 주소 지정 방식
jump 시에는 pc에 20비트 immediate만큼을 더한 주소를 사용함.
jump는 비교적 더 먼 위치로 이동하기 때문에 20비트만큼을 사용하는데, 이때의 20비트는 upper immediate에서의 20비트인 것은 아님. 20비트는 그냥 정해진 크기인 것.

더 먼 거리로 분기해야 하는 경우에 upper immediate를 써서 총 32비트 immediate를 사용함. 이때 lui로 12~31비트를 지정하고 jalr로 0~11비트를 지정해 분기함.

3.5. Pseudo-instructions

3.5.1. Pseudo-instructions

Definition 41 *실제로 구현되어 있는 연산은 아니지만, 어셈블리어를 보는 사람의 편의를 위해서 RV32I에 정의되어 있는 연산들을 Pseudo-instructions라고 함.*

사용자에게는 Pseudo-instruction으로 보이지만, 실제로는 대응되는 다른 연산으로 구현이 됨.

Pseudo-instructions로는 아래와 같은 것들이 있음.

Pseudo-instruction	Base instruction(s)	Meaning
`li rd, imm`	`addi rd, x0, imm`	Load immediate
`la rd, symbol`	`auipc rd, D[31:12]+D[11]` `addi rd, rd, D[11:0]`	Load absolute address where D = symbol – pc
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`bgt{u} rs, rt, offset`	`blt{u} rt, rs, offset`	Branch if > (u: unsigned)
`ble{u} rs, rt, offset`	`bge{u} rt, rs, offset`	Branch if ≥ (u: unsigned)
`b{eq|ne}z rs, offset`	`b{eq|ne} rs, x0, offset`	Branch if { = | ≠ }
`b{ge|lt}z rs, offset`	`b{ge|lt} rs, x0, offset`	Branch if { ≥ | < }
`b{le|gt}z rs, offset`	`b{ge|lt} x0, rs, offset`	Branch if { ≤ | > }
`j offset`	`jal x0, offset`	Unconditional jump
`call offset`	`jal ra, offset`	Call subroutine (near)
`ret`	`jalr x0, 0(ra)`	Return from subroutine
`nop`	`addi x0, x0, 0`	No operation

4. Instruction format

4.1. Immediate의 저장과 적용

4.1.1. Immediate의 저장

format type별로 immeidate가 저장되는 형태가 다름. 이때 immediate는 signed 데이터로, 저장 시에 남은 공간은 sign extension됨. 참고로, imm가 12비트보다 커지면, cpu는 두 개의 명령어로 쪼갬. 물론 unsigned에 대해서는 0으로 extension됨.

1. I, S, U type
immediate값으로는 짝수/홀수 모두 사용되고, 어셈블리 명령어에 작성된 immediate 값 그대로가 메모리에 저장됨.

store/load에서도 메모리에 접근하지만 명령어들에 대해 접근하는 것이 아니므로 alignmnet를 신경쓸 필요가 없음. 2 byte로 alignment되어 있는 것은 명령어들임.

2. SB, UJ type
immediate값은 짝수만이 사용되고(2 byte alignment), 그에 따라 0번째 비트는 항상 1이므로 생략하여 저장함. 이때 저장 방식은 format type에 따라 상이함.

4.1.2. Immediate의 적용

instruction에 저장된 immediate값은 사용 시점에 레지스터에 저장되는데(자세한 것은 더 찾아보자), 이때 저장한 방식에 따라 값이 사용됨.

1. I, S, U type
저장된 immediate값을 signed extension한 값이 그대로 사용됨.

I type인 jalr은 특정 주소로 분기하기 때문에 명령어에 작성된 imm12와 레지스터의 값을 더한 것의 0번째 비트를 0으로 바꾼 후 반영됨. jalr x0, 0(ra)등의 사용에서는 분기하는 주소가 짝수이지만 일반적인 경우에서는 주소 값으로 홀수가 지정될 수 있기 때문임.

예를 들어, 명령어에 imm으로 작성되어 있으면 pc=(R[rs1] + SignExt(imm)) & (~1)과 같이 반영됨.

2. SB, UJ type
저장된 immediate값을 1만큼 왼쪽으로 shift하고, 남은 비트는 signed extension한 값이 사용됨. 즉, 어셈블리 명령어에 작성된 값이 그대로 반영되는 것.

예를 들어, 명령어에 imm으로 작성되어 있으면 PC = PC + SignExt(imm[12:1] « 1)와 같이 반영됨.

4.2. Instruction format

4.2.1. Instruction format

Definition 42 *RV32I 모든 명령어들의 크기는 32비트(4바이트)이고 format 종류(type)에 따라 나름의 형태를 가지고 있음.*

RV32I에는 R, I, S, SB, U, UJ로 총 6개의 format type이 존재함. cpu는 명령어의 funct7, funct3, opcode로 어떤 type의 어떤 연산인지를 판단함. funct3와 funct7은 additonal opcode임.

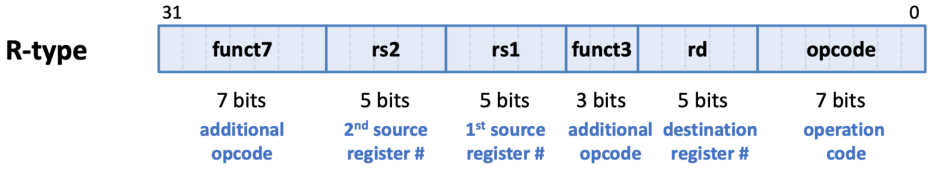
immediate가 쪼개져 있는 등 각 type의 구성 요소들이 독특한 배열을 가지는 것은 다른 type들과 값의 위치를 맞추기 위해서임. 이를 통해 하드웨어적인 wire를 간단히 연결할 수 있음.

4.2.2. R-type

Definition 43 *R-type(Register)은 두 개의 source register와 하나의 destination register가 존재하는 연산들의 format type임.*

산술 연산, 논리 연산에 대한 format type임.

아래와 같은 format을 가짐.



Instruction	Type	Example	funct7	funct3	opcode
add	R	add rd, rs1, rs2	0000000	000	0110011
sub	R	sub rd, rs1, rs2	0100000	000	0110011
sll	R	sll rd, rs1, rs2	0000000	001	0110011
slt	R	slt rd, rs1, rs2	0000000	010	0110011
sltu	R	sltu rd, rs1, rs2	0000000	011	0110011
xor	R	xor rd, rs1, rs2	0000000	100	0110011
srl	R	srl rd, rs1, rs2	0000000	101	0110011
sra	R	sra rd, rs1, rs2	0100000	101	0110011
or	R	or rd, rs1, rs2	0000000	110	0110011
and	R	and rd, rs1, rs2	0000000	111	0110011

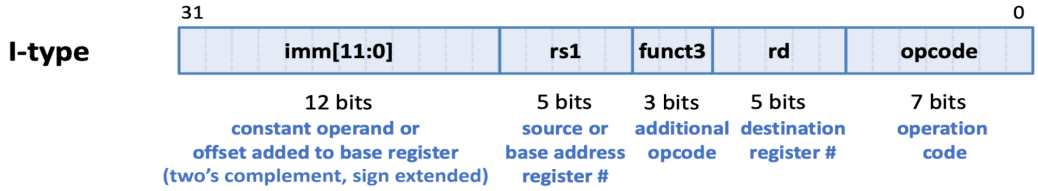
4.2.3. I-type

Definition 44 *I-type(Immediate)은 하나의 source register와 하나의 destination register, 12비트 immediate가 존재하는 연산들의 format type임.*

피연산자나 offset으로 12비트 immediate를 사용하는 산술 연산, 논리 연산, load 연산, jalr에 대한 format type임.

이때 shift 연산(slli, srli, srai)에 대해서는 funct7이 000000 shamt라고 적혀 있는데, 이때 shamt는 shift amount를 말함. 이는 imm[11:0]값의 첫 6비트는 funct7이 들어가고 나머지는 shift할 양(shamt)이 들어간다는 것임. 한 번에 shift할 수 있는 양은 프로세서의 shift logic에 의해 결정되는데 이 경우 이를 최대 6비트로 제한을 둔 것임.

아래와 같은 format을 가짐.



Instruction	Type	Example	funct7	funct3	opcode
addi	I	addi rd, rs1, imm12	-	000	0010011
slti	I	slti rd, rs1, imm12	-	010	0010011
sltiu	I	sltiu rd, rs1, imm12	-	011	0010011
xori	I	xori rd, rs1, imm12	-	100	0010011
ori	I	ori rd, rs1, imm12	-	110	0010011
andi	I	andi rd, rs1, imm12	-	111	0010011
slli	I	slli rd, rs1, shamt	000000 shamt	001	0010011
srli	I	srli rd, rs1, shamt	000000 shamt	101	0010011
srai	I	srai rd, rs1, shamt	010000 shamt	101	0010011
lb	I	lb rd, imm12(rs1)	-	000	0000011
lh	I	lh rd, imm12(rs1)	-	001	0000011
lw	I	lw rd, imm12(rs1)	-	010	0000011
ld	I	ld rd, imm12(rs1)	-	011	0000011
lbu	I	lbu rd, imm12(rs1)	-	100	0000011
lhu	I	lhu rd, imm12(rs1)	-	101	0000011
lwu	I	lwu rd, imm12(rs1)	-	110	0000011
jalr	I	jalr rd, imm12(rs1)	-	000	1100111

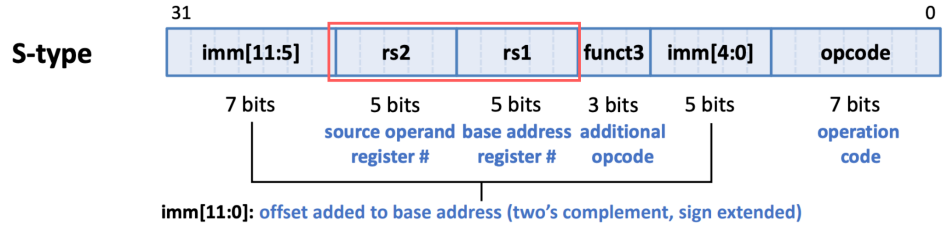
4.2.4. S-type

Definition 45 *S-type(Store)은 두 개의 source register와 12비트 immediate가 존재하는 연산들의 format type임.*

offset으로 12비트 immediate를 사용하는 store 연산에 대한 format type임.

I에서는 r1, rd, imm12을 사용했고, S에서는 r1, r2, imm12를 사용함. 다른 format들과의 r1, r2, rd 위치를 맞추기 위해 format type을 분리한 것.

아래와 같은 format을 가짐.



- Different immediate format for store instructions
  - Split so that rs1 and rs2 fields always in the same place

Instruction	Type	Example	funct7	funct3	opcode
sb	S	sb rs2, imm12(rs1)	-	000	0100011
sh	S	sh rs2, imm12(rs1)	-	001	0100011
sw	S	sw rs2, imm12(rs1)	-	010	0100011
sd	S	sd rs2, imm12(rs1)	-	011	0100011

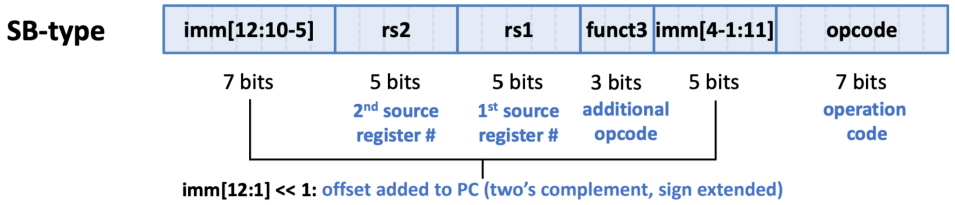
4.2.5. SB-type

Definition 46 *SB-type 또는 B-type(Branch)은 두 개의 source register와 12비트 immediate가 존재하고, 12비트 immediate가 항상 짝수인 연산들의 format type임.*

분기할 위치를 짝수 12비트 immediate로 지정하는 조건부 분기 연산(branch)에 대한 format type임.

명령어 상에서 지정한 immediate가 짝수이므로 0번째 비트를 생략한 형태로 저장됨. 기존의 12비트 immediate(0 11번째 존재)에서 13비트까지 Sign Extension을 하고, (부호 비트(MSB)에 해당하는) 11번째 비트를 0번째 비트 위치에 넣은 형태로 저장함.

아래와 같은 format을 가짐.



Instruction	Type	Example	funct7	funct3	opcode
beq	SB	beq rs1, rs2, imm12	-	000	1100011
bne	SB	bne rs1, rs2, imm12	-	001	1100011
blt	SB	blt rs1, rs2, imm12	-	100	1100011
bge	SB	bge rs1, rs2, imm12	-	101	1100011
bltu	SB	bltu rs1, rs2, imm12	-	110	1100011
bgeu	SB	bgeu rs1, rs2, imm12	-	111	1100011

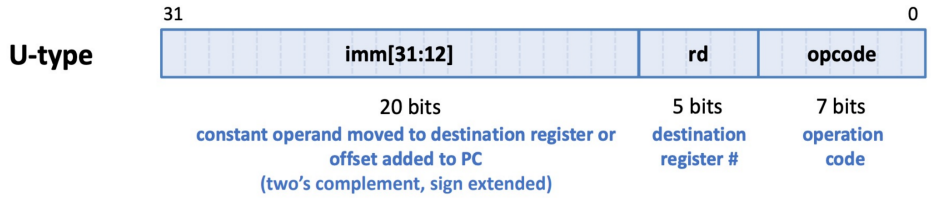
예를 들어 1111 1101 0010(이진수)이라는 값을 immeidate로 SB-type 명령어에 저장할 때는, 1111 1101 0011이 7비트, 5비트로 나뉘어 들어감.

4.2.6. U-type

Definition 47 *U-type(Upper)은 하나의 destination register와 20비트 immediate가 존재하는 연산들의 format type임.*

20비트 upper immediate를 지정하는 lui, auipc 연산에 대한 format type임.

아래와 같은 format을 가짐.



- 20-bit immediate is shifted left by 12 bits

Instruction	Type	Example	funct7	funct3	opcode
lui	U	lui rd, imm20	-	-	0110111
auipc	U	auipc rd, imm20	-	-	0010111

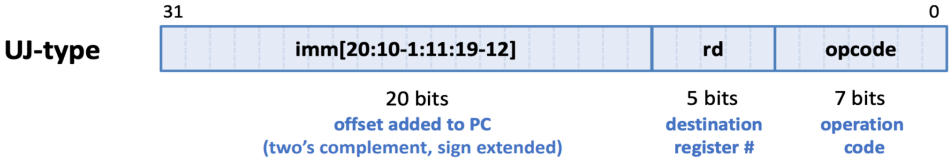
4.2.7. UJ-type

Definition 48 *UJ-type(Unconditional Jump)은 하나의 destination register와 20비트 immediate가 존재하고, 20비트 immediate가 항상 짝수인 연산들의 format type임.*

분기할 위치를 짝수 20비트 immediate로 지정하는 jal에 대한 format type임.

명령어 상에서 지정한 immediate가 짝수이므로 0번째 비트를 생략한 형태로 저장됨. 기존의 20비트 immediate(0 19번째 존재)에서 21비트까지 Sign Extension을 한 데이터를 넣음. 부호 비트(MSB)에 해당하는 값을 19번째 비트 위치(맨 앞)에 넣고, 수에 해당하는 부분을 그 다음부터 작성하고, 부호 비트에 해당하는 비트를 바로 다음 비트에 작성하고, 남은 비트들은 부호비트를 Sign Extension한 값으로 채움.

아래와 같은 format을 가짐.



- 20-bit immediate is shifted left by 1 bit and added to PC

Instruction	Type	Example	funct7	funct3	opcode
jal	UJ	jal rd, imm20	-	-	1101111

예를 들어, 0111 1101 0000(이진수)은 0111 1101 0000 0000 0000으로 저장됨.

5. 조건문과 반복문

조건문/반복문 등이 RISC-V에서 어떤 어셈블리로 어떻게 구성되어 있는지를 정리함.

기존의 코드를 goto를 사용하는 것으로 바꾸어 놓고 생각하면 더 편함.

5.1. 조건문

5.1.1. if문

조건이 참인 경우 코드를 건너뛰도록 분기하고, 조건에 맞지 않은 경우 그대로 수행하다가 조건이 참인 경우 분기되는 위치를 만나기 전에 무조건 분기가 되게 함.

이때 무조건 분기는 무조건 분기 연산으로 할 수도 있고, 항상 참인 조건 분기를 사용할 수도 있음.

아래는 그 예시임.

```
// c 코드
if(i == j) f = g + h;
else f = g - h;

// RISC-V 어셈블리어
// i-x22, j-x23, f-x19, g-x20, h-x21
        bne x22, x23, L1    // 비교.
        add x19, x20, x21   // 참인 경우. f = g + h
        beq x0, x0, EXIT    // 참인 경우. 무조건 분기가 됨.
L1:     sub x19, x20, x21   // 거짓인 경우. f = g - h
EXIT:   ...
```

5.2. 반복문

5.2.1. while문

조건을 검사하여 참이면 Loop label로 분기하고, 거짓이면 Exit label로 분기하도록 함.

while문은 아래의 형태를 가짐.

```
Loop:   참이면 진행, 거짓이면 goto Exit
        body
        goto Loop
Exit:   ...
```

아래는 그 예시임.

```
// c 코드
while(A[i] == k)
{
    i += 1;
}

// RISC-V 어셈블리어
// i-x32, k-x24, A[]-x25, A의 각 요소는 4바이트
Loop:   slli x10, x32, 2    // 조건 검사. i * 4. A의 요소 크기만큼 곱해줌.
        add x10, x10, x25   // 조건 검사. 가리킬 주소 지정.
        lw x9, 0(x10)       // 조건 검사. load.
        bne x9, x24, Exit   // 조건 검사.
        addi x32, x32, 1    // 작업.
        beq x0, x0, Loop    // Loop로 분기.
Exit:   ...
```

5.2.2. do-while문

while과 동일한데, 조건 검사를 맨 마지막에 하면 됨.

do-while문은 아래의 형태를 가짐.

```
Loop:   Body
        참이면 goto Loop, 거짓이면 그대로 진행
        ...
```

5.2.3. 반복문 변환

컴파일러에 따라 어셈블리를 생성하는 방법에는 차이가 존재하는데, 아래와 같이 for->while->do-while->goto로 변환하여 생각하는 방법도 존재함. 당연하게도 모든 컴파일러에서 모든 경우에 이런 것은 아니므로, 이런 것도 있다 알고 넘어가자.

RISC-V에서도 Og 옵션에서는 while을 그대로 생성하지만, O2 옵션에서는 while문을 do-while문으로 변환해 생성하기도 함. 이 경우 더 성능이 뛰어나도록 컴파일한 수도 있음.

아래와 같이 변환이 됨.

```
// for문
for(init; test; update)
    body;

// while문
init;
while(test)
{
    body;
    update;
}

// do-while문
    init;
    if(!test) goto Exit;
    do
    {
        body;
        update;
    }while(test)
Exit:

// goto문
    init;
    if(!test) goto Exit;
Loop:
    body;
    update;
    if(test) goto Loop;
Exit:
```

6. 프로시저

6.1. 프로시저

6.1.1. 프로시저

Definition 49 *특정 작업을 위해 사용되는 일련의 절차나 코드 블럭을 프로시저(procedure)라고 함.*

다른 프로시저를 호출한 프로시저를 Caller, 호출당한 프로시저를 Callee라고 함. 또한, 다른 프로시저의 호출을 포함하는 프로시저를 non-leaf 프로시저, 그렇지 않은 프로시저를 leaf 프로시저라고 함.

엄밀히 말하면 함수와 프로시저는 다르지만, 유사한 것으로 생각할 수 있음.

6.1.2. RISC-V stack

Definition 50 *RISC-V에서는 프로시저 구현을 위해 메모리 공간에서 LIFO을 따르는 stack을 사용함.*

stack은 거꾸로 자람. 높은 주소를 기준으로 잡아 두고, 해당 주소부터 낮은 주소로 요소를 삽입함. 이때 가장 바깥쪽 요소(stack의 top)의 주소는 x2(sp)에 저장함.

RISC-V에는 push, pop 연산이 존재하지 않기 때문에 store, load로 스택에서 값을 넣고 꺼냄.

참고로, c/c++, pascal, java 등은 stack 기반 pl(programming language)임. 스택 기반 pl들은 아래의 특징을 가짐.

1. 하나의 프로시저를 여러 번 사용할 수 있음.
프로시저를 인스턴스화하여 재진입이 가능함.

2. 각 인스턴스에 대한 상태를 저장하기 위한 공간을 필요로 함.

3. 프레임으로 스택을 할당함.

6.1.3. stack frame

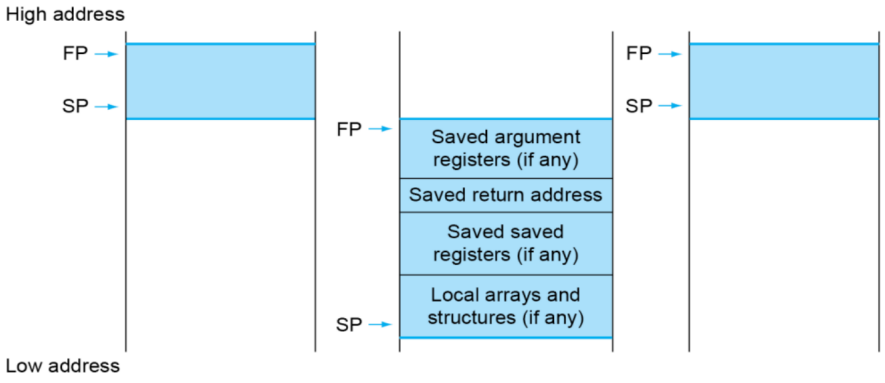
Definition 51 *프로시저에 대한 정보는 stack에 frame의 형태로 저장되는데, 이를 stack frame이라고 함.*

stack frame에는 Caller-saved 레지스터들(반환 주소, 인자, 지역변수(자동변수), 임시변수 등)이 저장됨.

frame의 가장 바깥쪽(낮은 주소)는 x2(sp)로 가리키고, optional하게 가장 바깥쪽 frame의 시작 주소를 x8(fp, frame pointer)에 저장하기도 함.

frame의 생성(할당)에 대한 코드를 setup code, frame의 삭제(반납)에 대한 코드를 finish code라고 함.

stack frame에 데이터를 저장해 둬야 프로시저의 계속된 호출에서도 값을 사용할 수 있음.



6.1.4. 인자 전달

인자 전달을 위한 레지스터는 8개 뿐이기 때문에, 이보다 인자가 많이 전달되면 남은 것들은 stack에 역순으로 넣음. 이때 역순으로 넣는 것은 꺼내는 순서를 고려한 것임.

예를 들어, arg0~arg9까지 10개의 인자를 전달하려고 하면, arg0~arg7은 a0~a7에 넣고, stack에는 arg9, arg8 순으로 넣음. 이후에 꺼낼 때 arg8, arg9 순으로 나오게 하기 위함임.

6.2. 프로시저의 구현

6.2.1. 프로시저의 구현

Definition 52 *어셈블리 수준에서 프로시저의 구현을 위해서는 아래의 항목들에 대한 구현이 필요함.*

1. Passing control : 제어의 이동. 함수 호출과 반환.
2. Passing data : 데이터 전달 방식. 인자 값의 전달과 반환값의 전달.
3. Memory management : 메모리 관리. 프로시저 실행 동안 할당하고 종료 시 반납.

이에 따라 leaf 프로시저 호출 시 수행되는 작업은 아래와 같음.

1. 프로시저의 인자(argument)는 x10~x17(a0~a7)에 저장함.
2. 프로시저의 수행 이후 수행할 명령의 주소를 x1(ra)에 저장하고 프로시저로 분기.
3. 프로시저 수행에 필요한 메모리 공간 확보.
4. Callee-saved register를 stack에 삽입/저장.
5. 프로시저의 명령 수행.
6. 반환값을 a0과 a1에 저장.
7. Callee-saved register를 stack에서 추출/복원.
8. x10(a1), x11(a2)에 반환값 지정.
9. x1(ra)에 저장된 주소로 이동해 명령 실행.

non-leaf 프로시저의 경우 다른 프로시저를 호출하기 전에 자신의 반환 정보, 인자, 지역변수(자동변수), 임시변수 등을 스택 프레임에 저장함. 이후 다른 프로시저의 수행이 종료되고 제어가 넘어오면 해당 값들을 스택 프레임에서 꺼냄.

6.2.2. 메모리 공간 확보

프로시저가 호출되면 스택 프레임에 저장할 데이터를 위한 메모리 확보를 함. 그 과정은 아래와 같음.

1. sp 값을 필요한 만큼 증가시킴.
2. stack에 레지스터 값을 store함.
3. 작업 수행.
4. stack에서 레지스터 값을 역순(LIFO)으로 load함.

아래는 그 예시임.

```
// c 코드
int leaf(int g, int h, int i, int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}

// RISC-V 어셈블리어
// g-x10, h-x11, i-x12, j-x13
leaf:
    addi sp, sp, -8     // 공간 확보
    sw x18, 4(sp)       // Callee-save register를 stack에 삽입
    sw x19, 0(sp)       // Callee-save register를 stack에 삽입
    add x18, x10, x11   // 연산 수행
    add x19, x12, x13   // 연산 수행
    sub x10, x18, x19   // 연산 수행, 반환값 지정
    lw x19, 0(sp)       // Callee-save register를 stack에서 추출
    lw x18, 4(sp)       // Callee-save register를 stack에서 추출
    subi sp, sp, 8      // 공간 반납
    jalr x0, 0(ra)      // 호출된 곳으로 제어 이동
```

6.2.3. register saving conventions

Definition 53 *특정 레지스터들은 그 종류에 따라 저장 규칙이 다름. Caller-saved 레지스터인지, Callee-saved 레지스터인지에 따라 Callee의 해당 레지스터 값 보존 필요 여부가 결정됨.*

1. Caller-saved 레지스터
: 호출 전에 Caller가 그 값을 저장하는 레지스터. a0∼a7, t0∼t6.
Caller가 저장했으므로 Callee의 작업에서 해당 값이 변경되어도 괜찮음.

2. Callee-saved 레지스터
: Callee가 그 값을 보존해야 하는 레지스터. s0∼s11.
Caller가 값을 따로 저장해두지 않았으므로 Callee가 해당 값을 보존해야 함. 해당 레지스터에 대한 값의 변경이 예상되면 Callee 작업 수행 전에 stack에 그 값을 저장해두고, 이후 반환 전에 원래대로 복구해야 함.

즉, s가 붙은(saved) s0∼s11의 값을 변경하게 되는 경우 stack에 저장했다가 작업 이후에 복원해야 함.

Part IV

processor

1. datapath

1.1. single-cycle processor

1.1.1. execution 과정

명령어 실행의 과정을 더 세분화하면 아래와 같음.

1. Instruction fetch : 메모리에서 processor로 명령어를 가져옴.
2. Instruction decode : 명령어를 디코딩하여 필요한 작업을 지정함.
3. Operand decode : 피연산자(operand)에 해당하는 값을 디코딩하여 가져옴.
4. Execute : 명령어에 해당하는 작업을 수행함.
5. Result store : 결과값을 저장함.
6. Instruction next : 다음으로 수행할 명령어를 정의함.

1.1.2. single-cycle processor

Definition 54 *single-cycle processor는 각 명령어의 실행에 하나의 clock cycle을 사용하는 processor임. 즉, 각 명령어들이 순차적으로 처리됨.*

1. single-cycle processor의 장점
로직/클록 등의 구현이 편리함.

2. single-cycle processor의 단점
single-cycle processor에서는 명령어가 전부 single-cycle에 수행되고, 그에 따라 clock cycle time이 가장 느린 명령어(critical path)에 의해 결정됨. 이 경우 수행 시간 측면에서 명령어 별 최적화가 되지 않음. R-type, load/store, beq 명령어만이 존재한다고 가정하면, load가 critical path가 됨.

명령어 별로 사용하는 요소도 있고 사용하지 않는 요소도 있는데, 이게 따른 최적화도 이뤄지지 않음. 즉, memory와 functional unit의 활용률이 떨어짐.

후술하겠지만 multi-cycle(pipeline) processor로 single-cycle processor의 단점을 보완할 수 있음. 명령어를 몇 가지 단계로 나누고, 각 단계별로 하나의 cycle을 가지도록 하는 것. 이 경우 clock cycle time이 짧아지고, 명령어 별로 다른 CPI를 가지게 됨. 다만, 이를 구현하려면 단계 사이마다 상태를 저장할 레지스터를 추가로 사용해야 함.

각 명령어가 read/write을 positive edge에만 수행할 수 있다면 1 cycle에 명령어 하나가 수행되기 어려우므로, 실제로는 positive edge와 negative edge 모두에서 read/write하는 등 다양한 기법을 사용해 명령어 당 1 cycle을 맞춤.

1.2. datapath

1.2.1. datapath

Definition 55 *processor(cpu) 내부에서 명령어를 실행하는 데 사용되는 데이터의 흐름과 제어 신호를 관리하는 processor(cpu)의 구성 요소를 데이터패스(datapath)라고 함.*

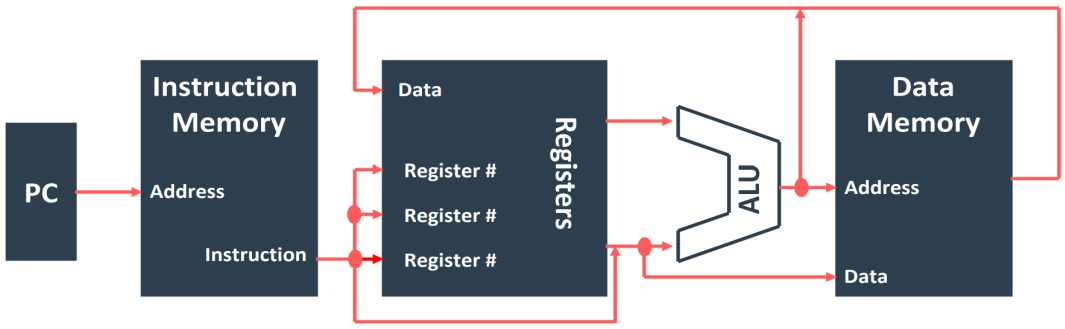
cpu는 datapath와 control로 구성된다고 생각할 수 있음. 물론 datapath에는 RAM 또한 있으므로 모든 부분이 cpu에 포함되는 것은 아님.

여기서는 단순한 구조에 대한 설명을 위해 RV32I의 명령어 중 일부(lw, sw, add, sub, and, or, beq)만을 다룸. 실제로 RV32I의 모든 명령어들을 전부 실행할 수 있는 datapath는 더 복잡한 구조를 가짐.

또한 단순함을 위해 우선 single-cycle processor의 datapath를 확인함.

1.2.2. datapath의 기본 구조

단순하게 본다면, datapath는 pc, instruction memory, register file, ALU, data memory로 구성됨.



1.3. 디지털 시스템의 구현

물리적인 회로와 신호 등을 통해 디지털 시스템을 구현할 수 있음. RV32I의 datapath에서 디지털 시스템을 구현하는 요소(컴포넌트, component)로는 조합 회로, 순차 회로, 클럭 신호가 있음.

1.3.1. 조합 회로

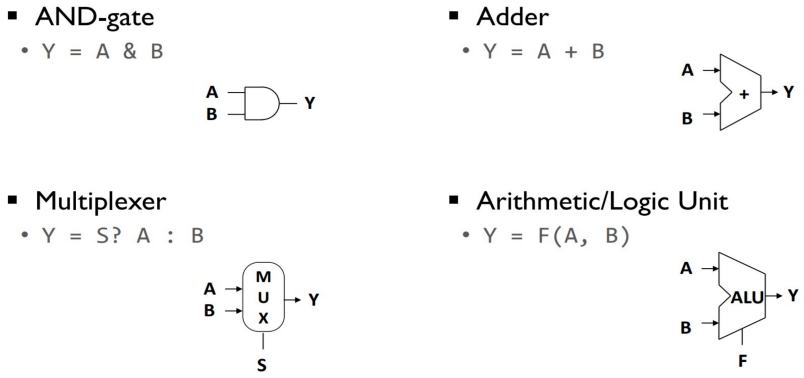
Definition 56 *조합 회로(combinational element)는 논리 게이트들의 비순환형(acyclic) 연결로, output이 오직 해당 연산에서의 input에 따라 결정되는 회로임.*

input이 들어오면 지정된 작업이 순차적으로 수행되기 때문에 내부 회로의 복잡도에 따라 어느 정도의 수행 시간이 소요됨(after some delay).

대표적으로 산술(arithmetic)/논리(logical) 연산이 조합 회로로 구성됨. 연산 종류에 따라 AND-gate, Adder, Multiplexer, ALU 등이 존재함.

조합 회로는 연산 종류에 따른 도미노처럼 동작함. 이런 논리 게이트들의 집합을 함수처럼 생각할 수도 있음.

아래와 같이 AND-gate, OR-gate, Adder, Multiplexer, Arithmetic/Logic Unit 등의 조합 회로들이 존재하고, 각 종류에 따른 모양으로 표기함.



1.3.2. 순차 회로

Definition 57 *순차 회로(sequential element)는 output이 현재 상태와 input에 의해 결정되는 회로임. 대표적으로 register에 비트 값을 저장(write)하는 회로로 사용됨.*

순차 회로는 신호에 의한 차단기처럼 동작함.

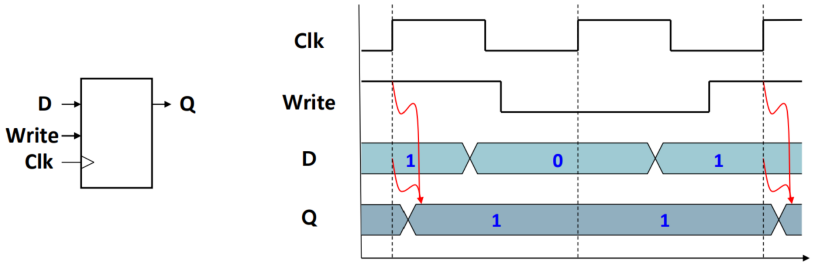
1.3.3. 레지스터에서의 순차 회로

레지스터에서는 clock 신호와 write 신호(control)에 의한 순차 회로로 데이터를 저장함.

레지스터에서 순차 회로를 사용하여 1비트를 저장하는 작업은 다음과 같이 수행됨. 우선 clock 신호는 심장 박동처럼 계속해서 일정한 주기로 반복됨. 이때 순차 회로는 positive edge마다 write 신호를 확인하는데, 1이면 input값을 ouput값으로 반영하고 0이면 작업을 수행하지 않음. 이때 신호가 물리적으로 회로를 거쳐야 하기 때문에 input값이 output값으로 반영되는 데에는 시간적 간격(delay)이 존재함.

실제로는 write 뿐만 아니라 read 또한 clock에 맞추어 동작함.

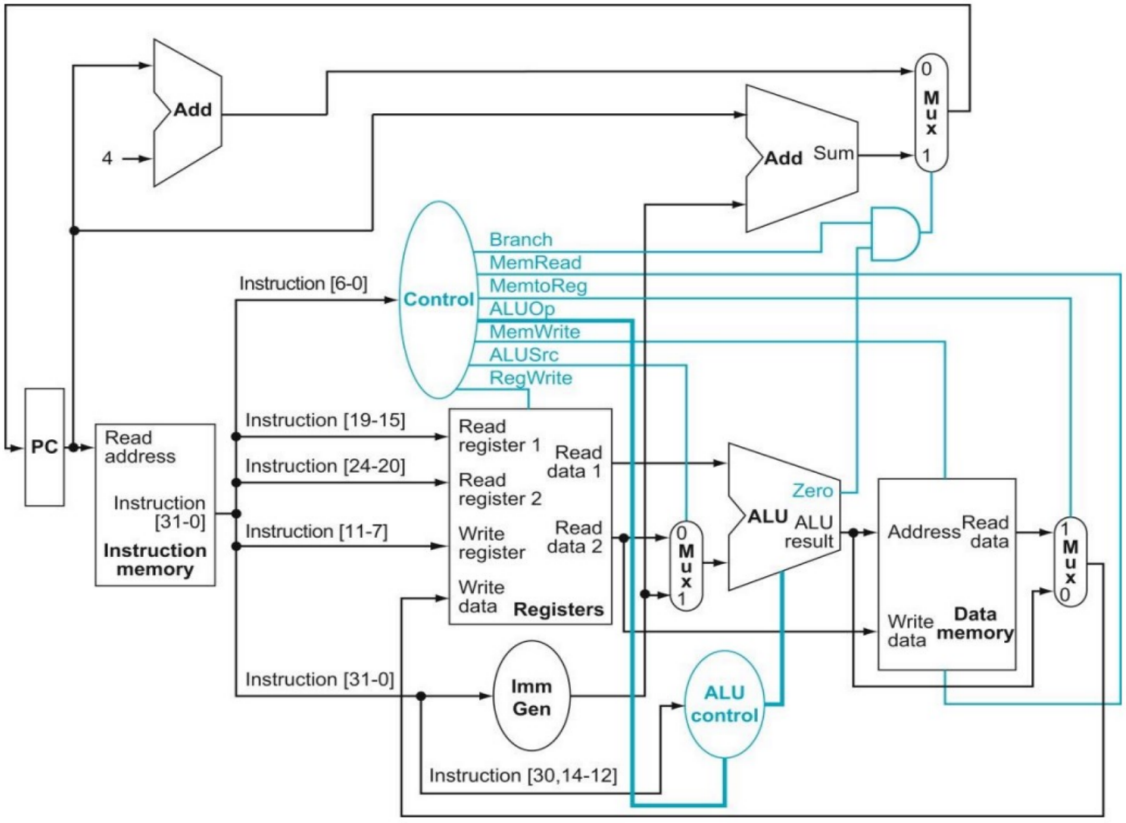
clock은 1과 0이라는 상태보다는 positive/negative edge가 중요함.



1.4. datapath의 구성요소

1.4.1. datapath의 구성요소

datapath는 아래와 같이 PC, ALU, register file, data memory, instruction memory, add/mux, imm gen 등으로 구성되어 있음. 아래의 그림은 datapath와 control을 함께 나타낸 것임.



1.4.2. ALU

Definition 58 *ALU(Arithmetic/Logic Unit)는 산술/논리 연산을 수행하는 게이트들의 집합임.*

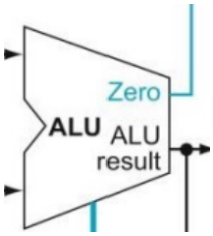
control의 ALUOp 신호와 명령어 종류에 따라 ALU control이 ALU에 신호를 보내면, 두 가지 피연산자에 대해 적절한 연산이 선택되어 수행됨.

1. ALU의 연산 수행
load/store의 경우 내부적으로 add 연산(rs1+imm12)이 수행되고, branch(beq)의 경우 내부적으로 sub 연산이 수행되고, R-type의 경우 해당 연산이 수행됨.

2. zero 신호
일반 산술/논리 연산의 경우 연산의 결과가 출력되지만, beq의 경우 내부적으로 sub 연산을 수행하여 그 값이 0이면(같은 경우) zero 신호로 1을, 0이 아니면(다른 경우) zero 신호를 0으로 전송함.

연산 결과가 0이면 zero 신호가 전송되는데, zero 신호와 branch 신호의 AND 연산을 수행하여 출력된 신호를 pc값 지정을 위한 mux에 사용함. 이 mux는 다음에 수행할 명령어에 대한 pc값으로 pc+4을 적용할 것인지, pc+offset을 적용할 것인지를 결정하는 회로임.

load/store, beq 또한 내부적으로는 add와 sub 연산만을 필요로 하기 때문에 ALU로 연산을 수행할 수 있음.



1.4.3. register file

Definition 59 *register file은 register들의 집합이고, 아래의 그림과 같은 구조를 가짐.*

1. read/write 포트
register file은 read/write을 위한 복수의 포트들을 가지기 때문에, 한 cycle 내에서 여러 개의 값을 읽고 쓸 수 있음.

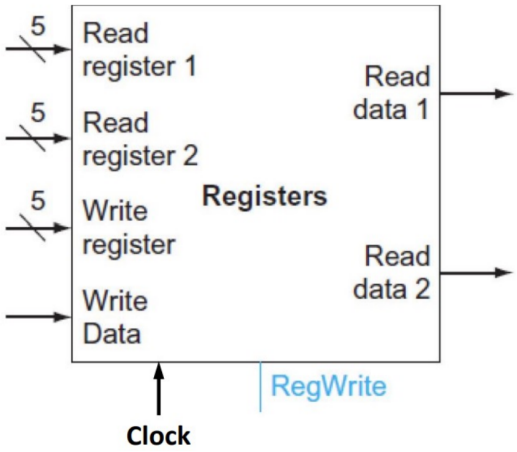
RV32I에서 register는 총 32개이므로 register를 지정하는 부분은 5비트 크기의 bus(wire)가 연결되어 있음. Read data 1,2와 Write data는 32비트 크기의 bus(wire)가 연결되어 있음. 추가로, 0값에는 x0(hard wired zero)가 사용됨.

register의 값을 내보낼 때 read register 1의 값은 read data 1으로, read register 2의 값은 read register 2로 보냄.

write register에는 write을 수행할 register를, write data에는 write할 값이 들어옴.

2. clock/RegWrite 신호
write시에는 순차 회로를 위한 clock 신호와 write 신호가 필요함. 이때 RegWrite 신호가 write 신호로서 기능함. RegWrite이 1이면 해당 register에 데이터가 저장(write)됨.

RegWrite이 0이면 read를 수행하는 것으로, output이 input(register 주소)에 의해 결정되는 조합 회로처럼 동작함. RegWrite이 1이면 write을 수행하는 것으로, 순차 회로로서 동작함.

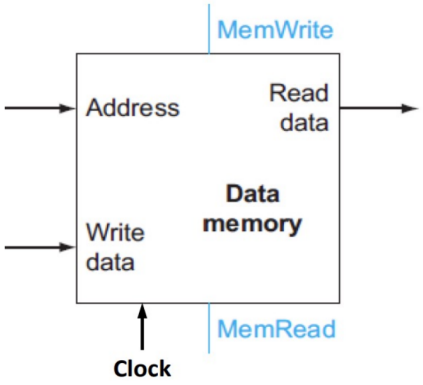


1.4.4. data memeory

Definition 60 *data memory는 RAM에서 프로그램 데이터를 저장하기 위한 부분임. 아래의 그림과 같은 구조를 가짐.*

register file과 유사한 구조를 가지며 유사하게 동작함. 즉, read 시에는 조합 회로처럼 동작하고 write 시에는 순차 회로로서 동작함. 단, data memory는 read와 write을 위한 MemRead와 MemWrite 신호를 모두 사용함.

일부 명령어(load)만이 data memory에서 데이터를 읽어오므로, 불필요한 read를 수행하지 않기 위해서 MemRead 신호 또한 존재하는 것임.

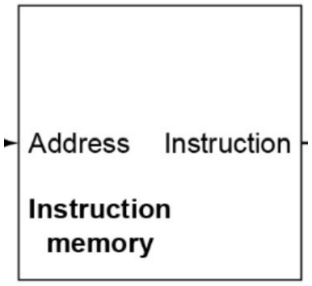


1.4.5. instruction memory

Definition 61 *insturction memory는 RAM에서 명령어를 저장하여 읽기 전용으로 사용하는 부분임.*

RAM을 data memory와 instruction memory로 구분한 것은, instruction을 read하는 작업이 반복해서 수행되어야 하기 때문임.

instruction memory는 read만 수행하므로 신호가 필요 없음.



1.4.6. multiplexer

Definition 62 *multiplexer(mux)는 여러 개의 input 신호들을 하나의 output 신호로 내보내는 조합 회로임.*

mux를 사용하여 여러 신호가 합쳐지는 부분에서 특정 신호를 선택하여 내보내는데, control의 select 신호로 어떤 신호를 내보낼지를 결정함. 이를 통해 명령어에 따라 다른 data source를 사용할 수 있도록 함.

mux에 2^n 개의 input line이 연결되어 있다면, 특정 신호를 선택하기 위해 n비트 크기의 select 신호가 필요함.

당연하게도 mux는 여러 신호를 병합할 때 사용하는 것이지, 하나의 신호를 여러 line으로 분리할 때는 사용하지 않음.

1.4.7. imm gen

Definition 63 *immediate generator unit은 명령어에서 immediate에 해당하는 부분을 가져와 작업에 사용할 원래의 immediate를 복원하여 출력하는 요소임.*

명령어 인코딩 시에 sign extend하여 순서를 뒤바꿔 놓은 것을 원래의 순서로 복원하고, 1만큼 shift left 하는 등의 과정을 수행함. 물론 jalr 등에서는 다른 형태의 과정을 가지겠지만, 여기서는 다루지 않음.

1.4.8. control

Definition 64 *control은 명령어에서 opcode를 받아, 해당 작업에 필요한 신호를 생성하여 datapath의 각 부분으로 보내는 요소임.*

control은 제어 신호의 유형을 분류해 놓고, mux에 신호를 보내 data 흐름을 제어하고, 각 요소의 상태를 지정하고, 명령어에 따른 제어를 수행함.

1. opcode
opcode는 명령어의 범주에 따라 지정되어 있고, funct3와 funct7으로 해당 범주 내에서 명령어를 구분함. 이때 이 범주는 format type이 아니라 산술/논리 연산, load 연산, store 연산, branch 연산 등 datapath에서 명령이 수행되는 형태에 따른 구분임.

그렇기 때문에 control은 opcode에 해당하는 instruction[6-0]으로 해당 명령어에 대한 신호를 각 요소에 전달하여 datapath를 제어함.

2. control의 신호 전송
single-cycle에서 control은 아래와 같은 신호들을 전송함.

RegWrite : register에서의 write(1)/read(0) 지정.
ALUSrc : ALU의 source로 들어갈 rs2(0)/imm(1)를 지정.
ALUOp : ALU에서 수행할 연산 지정.
MemWrite : data memory에서의 write(1) 지정.
MemRead : data memory에서의 read(1) 지정.
MemtoReg : register의 Write data로 들어갈 ALU의 연산 결과(0) 또는 data memory(1) 값을 지정.
PCSrc : pc를 update할 pc+4(0)/pc+imm(1)를 지정. branch와 zero에 대한 AND 연산으로 표기하기도 함.
branch : 현재 명령이 branch 명령어인지(1) 지정.

각 신호들 중에 해당 명령어에서 사용되지 않는 신호는 어떤 값을 가지더라도 상관 없기 때문에 don't care라고 부름. 이때 don't care는 0/1이 아니라 X임. 물론 don't care는 기본적으로 0으로 지정하여 안정성을 높임.

3. ALUOp
현재 상황에서 ALUOp는 (4가지 연산이므로) 아래와 같이 2비트 크기로 나타낼 수 있음. ALU contorl에는 funct3, funct7이 전달되므로 이후 구체적으로 연산이 지정됨.

00 : lw, sw (add 수행)
01 : beq
10 : R-type

1.4.9. ALU control

Definition 65 *ALU control은 control과 분리되어 ALU가 수행할 연산을 지정하는 요소임.*

1. ALU control의 신호 전송
ALU control은 control로부터 ALUOp 신호와, instruction의 [30, 14-12] 부분으로 어떤 연산을 수행할지를 결정하는 신호를 ALU에 보냄.

현재 상황에서 ALU에 전달되는 data는 아래과 같이 4비트 크기로 각 연산을 지정할 수 있음.

0000 : AND
0001 : OR
0010 : add
0011 : sub

2. ALUOp, funct3, funct7
ALU control은 ALUOp와 funct7, funct3에 해당하는 instruction[30, 14-12]를 읽어 ALU에 신호를 보냄. 이때 [14-12]가 funct3에 해당하는 부분이고 [30]이 funct7에 해당하는 부분임. 잘 보면 funct7은 하나의 비트 값만을 사용함.

1.5. 명령어 실행 과정

1.5.1. pc update(fetch)

pc값을 update할 때는 해당 명령어가 pc값에 +4를 하는 명령어인지, 다른 위치로 분기하는 명령어(branch)인지에 따라 작업이 수행되어야 함.

pc+4를 수행하는 Adder의 값과, pc+immediate(offset)을 수행하는 Adder의 값을 mux로 선택하여 사용함. 이때 mux의 신호로는 branch 신호와 zero 신호의 AND 연산으로 나온 신호를 사용하는데, pc+immediate(offset)을 적용하는 경우는 branch 명령어에 대해 조건 연산이 참인 경우이기 때문임.

1.5.2. R-type 명령어의 실행

R-type은 op rd, rs1, rs2로 구성되어 있으므로, 해당 피연산자들에 대해 적절한 연산이 수행되도록 회로를 생각할 수 있음.

1. pc값에 해당하는 명령어를 instruction memory에서 읽어옴.
2. 명령어의 각 부분이 적절한 요소로 전송됨.
3. register에서 read register 1,2, write register를 모두 입력받고, read data 1,2로 연산할 데이터들을 출력함.
4. rs2값이 연산에 사용되므로 ALUSrc 신호는 0임.
5. 해당 데이터들로 ALU에서 연산을 수행하고, 그 결과를 ALU result로 출력함.
6. MemWrite/MemRead 신호는 모두 0으로 data memory에 접근하지 않음.
7. MemtoReg 신호는 0으로 해당 연산 결과를 바로 register로 보냄.
8. 연산 결과를 write data로 받아서 write register에 해당하는 register에 write함. 이때 RegWrite 신호는 1임.
9. Branch/Zero 신호는 모두 0 이므로 pc값은 pc+4로 update됨.

1.5.3. load 명령어의 실행

load 연산(I-type)은 op rd, imm12(rs1)으로 구성되어 있으므로, 해당 피연산자들에 대해 적절한 연산이 수행되도록 회로를 생각할 수 있음.

1. pc값에 해당하는 명령어를 instruction memory에서 읽어옴.
2. 명령어의 각 부분이 적절한 요소로 전송됨.
3. RegWrite가 0으로, register에서 read register 1, write register를 입력받고, read data 1로 연산할 데이터를 출력함.
4. imm값과 연산을 하므로 ALUSrc는 1임.
5. 해당 데이터들로 ALU에서 연산을 수행하고, 그 결과를 ALU result로 출력함.
6. MemWrite 신호는 0, MemRead 신호는 1로, data memory에서 read한 값을 read data로 보냄.
7. MemtoReg 신호는 1로 data memory의 값을 register로 보냄.
8. RegWrite가 1로, data memory의 값을 write data로 받아서 write register에 해당하는 register에 write함.
9. Branch 신호가 0 이므로 pc값은 pc+4로 update됨.

1.5.4. store 명령어의 실행

store 연산(S-type)은 op rs2, imm12(rs1)으로 구성되어 있으므로, 해당 피연산자들에 대해 적절한 연산이 수행되도록 회로를 생각할 수 있음.

1. pc값에 해당하는 명령어를 instruction memory에서 읽어옴.
2. 명령어의 각 부분이 적절한 요소로 전송됨.
3. RegWrite가 0으로, register에서 read register 1,2를 입력받고, read data 1,2로 연산할 데이터를 출력함.
4. imm값과 연산을 하므로 ALUSrc는 1임.
5. 해당 데이터들로 ALU에서 연산을 수행하고, 그 결과를 ALU result로 출력함.
6. MemWrite 신호는 1, MemRead 신호는 0로, data memory의 Address에 해당하는 부분에 Write data값을 write함.
7. MemtoReg 신호는 사용되지 않음.
8. Branch 신호가 0 이므로 pc값은 pc+4로 update됨.

1.5.5. beq 명령어의 실행

beq 연산(SB-type)은 op rs1, rs2, imm12로 구성되어 있으므로, 해당 피연산자들에 대해 적절한 연산이 수행되도록 회로를 생각할 수 있음.

1. pc값에 해당하는 명령어를 instruction memory에서 읽어옴.
2. 명령어의 각 부분이 적절한 요소로 전송됨.
3. RegWrite가 0으로, register에서 read register 1,2를 입력받고, read data 1,2로 연산할 데이터를 출력함.
4. rs2값이 연산에 사용되므로 ALUSrc 신호는 0임.
5. 해당 데이터들로 ALU에서 sub 연산을 수행하고, 그 결과가 0이면(같으면) zero 신호를 1로, 0이 아니면(다르면) 0으로 함.
6. MemWrite/MemRead 신호는 모두 0으로 data memory에 접근하지 않음.
7. MemtoReg 신호는 사용되지 않음.
8. Branch 신호가 1이므로 zero가 1이면 pc값은 pc+imm12로 update되고, zero가 0이면 pc값은 pc+4로 update됨.

2. pipeline

2.1. pipeline

2.1.1. pipeline

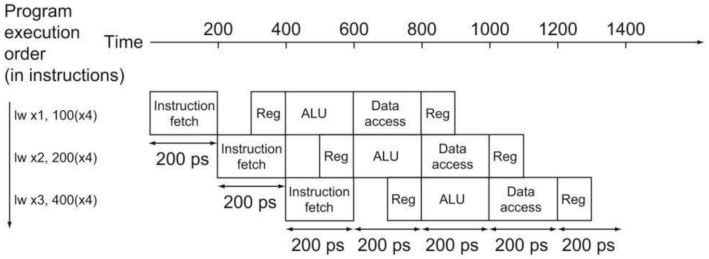
Definition 66 *pipelining은 명령어를 여러 stage로 나누어 datapath의 각 부분에서 해당되는 stage를 계속해서 수행하게 함으로써, 동시에 여러 명령어를 처리하는 것을 말함.*

명령어를 순차적으로 처리하는 single-cycle processor에서는 한 번에 하나의 명령어만을 처리할 수 있지만, pipelining을 적용하면 여러 명령어를 동시에 겹쳐서 처리할 수 있음.

당연하게도 pipelining을 통한 성능 향상은 명령어 당 처리 시간을 줄이지는 못하지만, 명령어들에 대한 처리량을 높인 것임.

pipelining은 코어 개수에 대한 이야기가 아니라, 코어 내부에서의 성능을 향상시키는 것.

물론 특정 stage가 필요 없어서 수행하지 않는 명령어에 대해서는 처리 시간이 줄었다고 할 수도 있지만, single-cycle에서도 clock cycle time이 가장 느린 명령어에 맞춰져 있으므로 오래 걸린 것이지 실제로 작업을 수행한 시간은 동일함.



2.1.2. stage

Definition 67 *명령어 처리 과정은 몇 가지 stage로 묶어서 분류할 수 있음. risc-V는 기본적으로 five stage pipeline을 사용함.*

1. IF(instruction fetch) : memory에서 instruction을 fetch.
2. ID(instruction decode) : instruction decoding, register read.
3. EX(execute) : execute/calculate.
4. MEM(memory) : access memeory.
5. WB(write back) : write result back to register.

2.1.3. pipeline 성능

Definition 68 *clock cycle time은 가장 느린 stage에 맞춰지기 때문에, stage들의 수행 시간이 균일할수록 pipeline의 성능이 좋음.*

stage별 수행 시간에 차이가 크다면, 어떤 stage가 계속 수행되고 있을 때 다른 stage에서는 미리 작업을 끝내고 아무 작업도 하지 않는 시간이 길어짐.

non-pipelined일 때의 실행 시간을 stage 개수로 나눈 것이 각 stage당 걸리는 시간이라면 최적임.

2.2. RISC-V pipeline

2.2.1. RISC-V으로의 pipelining

Definition 69 *RISC-V isa는 아래의 이유들 때문에 pipelining에 적합함.*

1. 명령어 길이가 32bit로 고정되어 있음.
IF, ID에서 수행해야 하는 작업량이 적어 1 cycle 안에 stage를 넣기 쉬움.
길이가 고정되어 있지 않다면 매번 검사하고 길이에 따른 작업을 수행해야 함.

2. 명령어의 format이 제한되어 있고 align되어 있음.
간단하게 설계할 수 있고 작업량이 적음.

3. 주소 지정 방식이 단순함.
주소 계산에 대한 추가적인 stage 대신 단순 ALU 연산으로 이뤄지는 것이 설계에 유리함.

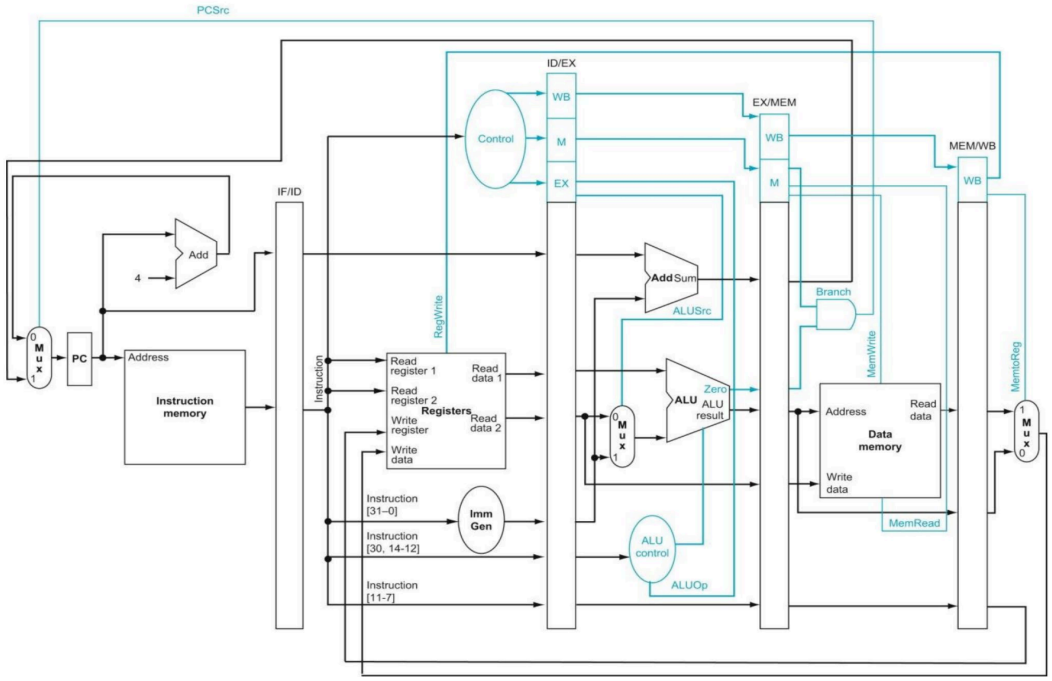
2.2.2. pipelined datapath

Definition 70 *pipelining을 하려면 각 stage 별로 다른 명령어에 대한 작업을 수행해야 하므로, stage 직전의 정보를 저장하는 register들이 추가로 필요함. pipeline에 들어가는 추가적인 register를 pipeline register라고 함.*

Write Register에 해당하는 값을 Register에 바로 넣으면 다른 명령어와 stage가 겹치게 되므로, Write Register 값을 register에 저장하며 WB까지 가져간 후에 전달해야 함.

control signal도 명령어 별로 존재하므로 register에 함께 저장하여 사용해야 함. stage별로 필요한 signal들이 저장되는데, 해당 stage가 끝나면 더 이상 저장하지 않아도 됨.

pipeline의 각 unit이 수행되는 clock에 의한 작업 순서는 학부 수준을 넘어서는 부분이므로 생략함. read-/write 등에 대해서는 single-cycle processor에서처럼 positive/negiative edge 모두에서 작업을 수행하는 등의 다양한 기법을 사용함.



2.2.3. hazard

Definition 71 *hazard는 다음 cycle에 수행될 명령어의 시작을 방해하는 상황으로, structural/data/-control hazard가 있음.*

각 hazard에 대한 해결 방법 또한 존재함.

pipelining을 적용하면 hazard가 발생할 수 있으므로 이에 대한 처리가 필요함.

2.3. structural hazard

2.3.1. structural hazard

Definition 72 *구조적 해저드(structural hazard)는 다른 명령어가 datapath의 리소스(ex. ALU, register)를 사용 중인 경우를 말함. 구조적인 문제 때문에 발생하는 hazard.*

structural hazard를 해결하는 방법으로는 아래와 같은 것이 있음.

1. pipeline stall(freezing)
2. 리소스 분리(data/instruction 등)

structural hazard를 방지하기 위해 L1 cache를 data cache와 instruction cache로 분리하고, register를 data register와 instruction register로 분리함. 분리되어 있지 않은 경우 포트가 각각 존재하는 것이 아니라면 IF, MEM stage가 겹치는 경우에도 structural hazard가 발생함.

지금 배우는 datapath의 memory는 data/instruction이 분리되어 있고, register의 경우에는 read/write 포트가 분리되어 있기 때문에 많은 문제가 발생하지는 않음.

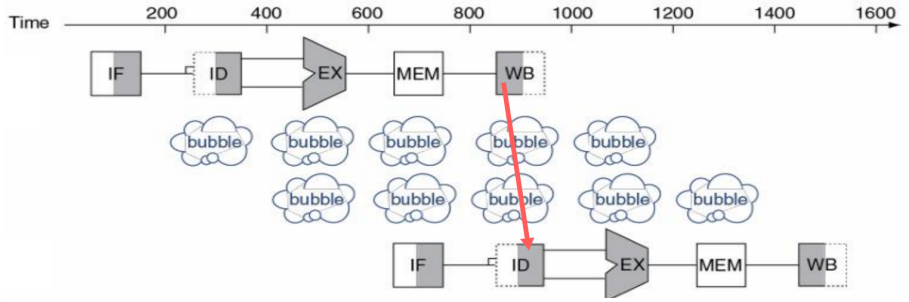
예를 들어, memory가 분리되어 있지 않은 경우 서로 다른 명령어의 MEM과 IF stage가 겹치는 경우가 있음.

2.3.2. stall

Definition 73 *pipeline을 특정 cycle만큼 정지시키는 것을 stall한다고 함. 이때 stall한 만큼 stage에 시간적으로 빈 공간이 생기는데 이를 bubble을 삽입했다고 함.*

삽입한 bubble의 개수만큼 이후 명령어의 수행이 미뤄짐.

모든 hazard는 stall로 해결할 수 있지만 pipeline을 멈추면 당연하게도 성능이 떨어짐. 가능하다면 다른 방법을 사용해야 함.



위 그림에서 IF는 IM, ID는 Reg, EX는 ALU, MEM은 DM, WB은 Reg로 나타내기도 함. 다이어그램에서의 포트 위치를 고려하여 오른쪽이 색칠된 것은 read하는 것, 왼쪽이 색칠된 것은 write하는 것을 의미함.

2.4. data hazard

2.4.1. data hazard

Definition 74 *데이터 해저드(data hazard)는 어떤 명령어의 수행이 이전 명령어의 데이터 사용에 영향을 받는 경우를 말함. 명령어 사이의 관계 때문에 발생하는 hazard. RAW(Read After Write) hazard라고도 함.*

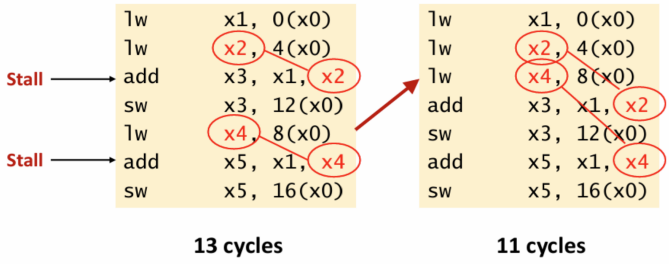
data hazard를 해결하는 방법으로는 아래와 같은 것이 있음.

1. *pipeline stall(freezing)*
2. *forwarding*
3. *compiler scheduling*
4. *Out-Of-Order execution(OoO Execution)*

예를 들어, 다른 명령어의 데이터 read/write 작업을 기다려야 하는 경우나 다른 명령어의 연산 결과를 사용해야 하는 경우가 있음.

2.4.2. compiler scheduling

Definition 75 *compiler scheduling은 data hazard가 발생하는 두 명령어 사이에 이 명령어들과 관련 없는 명령어를 끼워 넣어 처리하는 것을 말함. 즉, 컴파일러(소프트웨어)가 서로 관련없는 명령어들에 대해 실행 순서를 바꾸는 것.*



2.4.3. OoO execution

Definition 76 *비순차적 실행(OoO execution, Out-Of-Order execution)은 cpu가 명령어 실행 순서를 바꾸어 처리하는 것을 말함.*

compiler scheduling이 소프트웨어 수준에서 명령어 실행 순서를 바꾸는 것이라면, OoO execution은 하드웨어 수준에서 명령어 실행 순서를 바꾸는 것임.

이에 대해서는 더 자세히 다루지 않음.

현재 존재하는 대부분의 processor들은 OoO execution을 핵심적인 hazard 처리 방법으로 사용함.

2.4.4. forwarding

Definition 77 *포워딩(forwarding, bypassing)은 별도의 datapath를 만들어서 data를 전달하는 것을 말함.*

기존에는 이전 명령어가 연산을 마치고 register에 값을 write한 후 그 값을 사용해야 했다면, 두 명령어를 잇는 별도의 datapath를 사용해서 연산이 끝나자마자 다음 명령어로 데이터를 전달하는 것임.

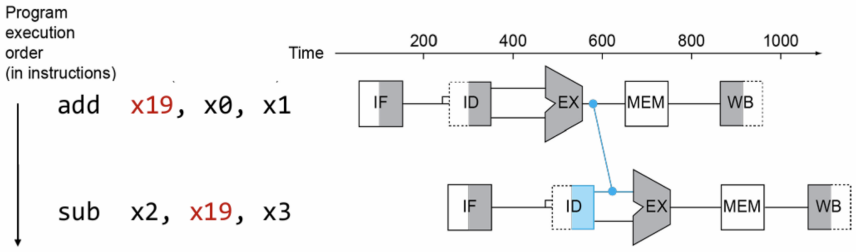
1. forwarding과 stall
forwarding으로 stall을 완전히 피할 수 있는 것은 아님. MEM 단계 이후에 값을 전송할 수 있는 등의 상황에서는 최소한의 stall이 필요함.

2. multiple forwarding
하나의 register 값을 이후에 처리되는 여러 명령어에서 사용해야 하는 경우(multiple readers), 한 명령어의 값이 여러 번 forwarding될 수 있음.

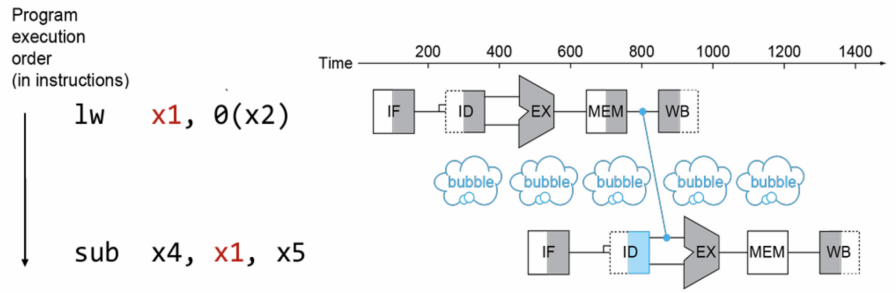
하나의 register에 여러 명령어에 대해 값이 write되는 경우(multiple writers), forwarding되는 데이터는 가장 최근에 연산된 데이터임.

당연하게도 forwarding 시에 시간을 역행할 수는 없음. 가로축이 시간일 때 왼쪽으로 datapath가 이어질 수 없음.

이전 명령어가 R-type이었다면 EX stage 이후 전달하면 되므로 forwarding으로 stall 없이 다음 명령어를 처리할 수 있음.



이전 명령어가 load였다면 MEM 단계 이후에 데이터가 존재하므로 forwarding으로 stall을 완전히 피할 수는 없음. 이런 경우를 특별히 Load-Use hazard라고 함.



2.5. hazard detection

어떤 경우에 hazard가 발생한 것으로 판정하여 forwarding을 적용할 것인지를 정의해야 함. pipeline에서 hazard와 forwarding에 대해 logic이 어떻게 하드웨어적으로 자동화되어 있는지를 알아봄.

2.5.1. forwarding logic

Definition 78 *hazard의 발생을 감지해 forwarding의 적용 여부를 판단하는 logic은 아래와 같이 hazard의 유형별로 구성되어 있음.*

이때 forwarding하는 이전 명령어가 register에 값을 write하는 명령어야 하고, 이 명령어의 rd가 x0(hard-wired zero)이 아니어야 함. x0이면 굳이 forwarding할 필요가 없음.

아래의 수식에서 ID/EX.RegisterRs1과 같이 작성된 것은 ID/EX register에 존재하는 rs1 값을 말함.

조건 충족 시에 전송되는 signal은 전달되는 데이터가 rs1으로 가는지, rs2으로 가는지에 따라 지정됨.

1. EX hazard
: ALU 연산 시에 rs1/rs2로 이전 명령어의 ALU 연산 결과인 rd값이 사용되어야 하는 경우, EX/MEM에서 ID/EX로 데이터를 넘겨줘야 하는 상황.

```
if(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRs1))
    forwardA = 10   // forwarding 적용 signal 지정

if(EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
    and (EX/MEM.RegisterRd == ID/EX.RegisterRs2))
    forwardB = 10
```

2. MEM hazard
: ALU 연산 시에 rs1/rs2로 이전 명령어가 ALU 연산 결과인 rd값이 사용되어야 하는 경우, MEM/WB에서 ID/EX로 데이터를 넘겨줘야 하는 상황.

EX hazard와의 차이점은 rd값을 전달하는 시점임.

EX hazard가 연속해서 발생하는 경우 MEM hazard의 logic을 EX hazard와 동일하게 지정하면 EX hazard가 MEM hazard로 처리될 수 있음. 즉, 최신 rd값이 반영되지 못할 수 있음. EX hazard가 발생하지 않는 경우에만 signal을 지정하도록 조건을 추가해야 함.

```
if(MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0) // EX hazard 제외
        and (EX/MEM.RegisterRd == ID/EX.RegisterRs1))
    and (MEM/WB.RegisterRd == ID/EX.RegisterRs1))
    forwardA = 01

if(MEM/WB.RegWrite and (MEM/WB.RegisterRd != 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
        and (EX/MEM.RegisterRd == ID/EX.RegisterRs1))
    and (MEM/WB.RegisterRd == ID/EX.RegisterRs2))
    forwardB = 01
```

3. load-use hazard
: ALU 연산 시에 rs1/rs2로 직전 load 명령어가 memory에서 read한 값을 사용되어야 하는 경우, 1번의 stall 이후 MEM/WB에서 ID/EX로 데이터를 넘겨줘야 하는 상황. stall만 넣어주면 MEM hazard의 hazard detection에 의해 데이터는 전송됨.

IF/ID와 ID/EX를 비교하여 판정하고 stall을 넣을 수 있음.

```
if(ID/EX.MemRead and ((ID/EX.RegisterRd == IF/ID.RegisterRs1)
    or (ID/EX.RegisterRd == IF/ID.RegisterRs2)))
    // load-use hazard로 판단, 1 cycle stall 삽입
```

시간선에 각 stage를 표시해 보면, MEM hazard의 EX hazard 제외 logic이 들어가지 않은 경우 EX hazard가 연속적으로 등장하면 두 cycle 후 EX hazard와 MEM hazard가 동시에 적용된다는 것을 확인할 수 있음. 가장 최근 값인 EX hazard만이 반영되도록 해 줘야 함.

본 logic을 사용하면 모든 forwarding이 필요한 부분이 처리됨. forwarding이 적용되어야 하는 부분을 정리하면 아래와 같이 3가지임.

1. load-use hazard인 경우 (1 stall)
2. R-R이 바로 붙어있는 경우 (0 stall)
3. R-R이 한 cycle 간격을 띄우고 붙어있는 경우. (0 stall)

두 칸 떨어진 경우는 Reg 단계가 겹치는데, 아마 detection에 따로 없는 것을 보아하니 쓰자마자 읽는 게 가능한 것으로 보임.

2.5.2. stall logic 구현

Definition 79 *load-use hazard의 stall 시에 수행되는 작업은 아래와 같음.*

1. ID/EX register의 control signal을 0으로 지정함.
EX stage는 nop가 됨.

2. IF, ID register의 값 전달과 pc update를 막음.
IF, ID에서는 직전에 수행한 작업을 한 번 더 수행하게 됨.

아무 작업도 수행하지 않는 것을 nop(no-operation)라고 함.

stall은 성능을 저하시키지만 정확한 결과를 위해 피할 수 없는 경우도 있는 것.

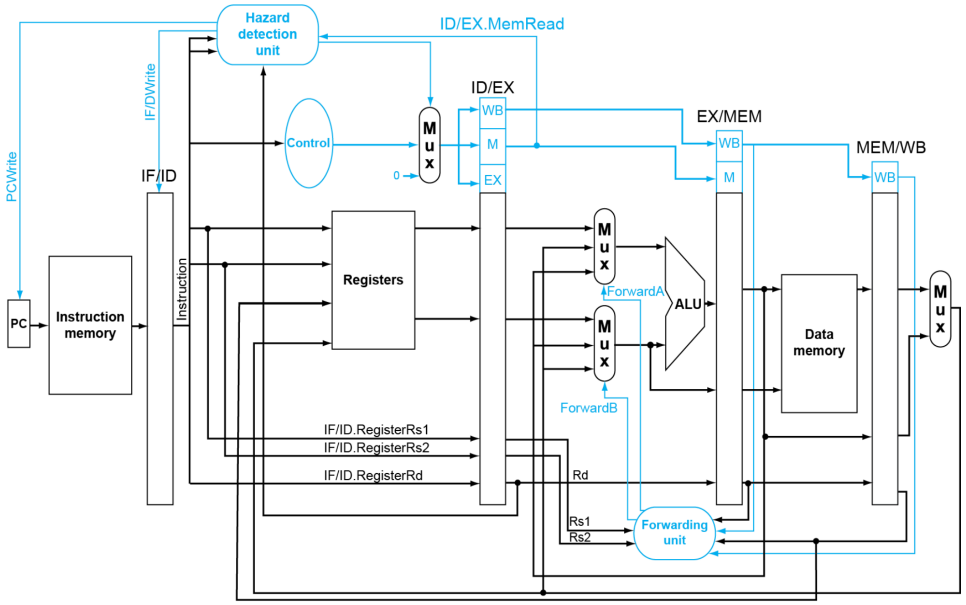
2.5.3. forwarding이 추가된 datapath

Forwarding unit을 추가하여 해당 unit에서 내부적으로 logic을 수행한 후 그 결과에 의해 signal을 전송함.

signal이 00이면 forwarding을 하지 않는 것, 01이면 MEM/WB register에서 값을 가져오는 것, 10이면 EX/MEM register에서 값을 가져오는 것임.

forwardA에 signal을 전송했으면 rs1에 값을 전달하는 것, forwardB에 signal을 전송했으면 rs2에 값을 전달하는 것임.

load-use hazard에 의한 stall을 하기 위해서 Hazard detection unit을 사용함. 해당 unit이 load-use hazard인 경우를 확인해 stall을 적용함. 이후 forwarding은 forwarding unit에 의해 처리됨.



물론 여기에 immediate 처리 mux 등을 추가할 수 있음.

2.6. control hazard

2.6.1. control hazard

Definition 80 *컨트롤 해저드(control hazard)는 다른 명령어에 의해 control action이 달라지는 경우를 말함.*

분기(ex. beq) 등으로 인해 제어가 넘어가는 경우, 바로 다음 명령어를 fetch하는 것이 아니라 분기하는 지점의 명령어를 fetch해야 하는데 별다른 처리가 없으면 계산기 끝나기 전에 이미 바로 다음 명령어들이 fetch됨.

control hazard를 해결하는 방법으로는 아래와 같은 것들이 있음.

1. pipeline stall(freezing)
2. branch prediction
3. compiler scheduling

branch이면 EX에서 분기 연산이 끝날 때까지 stall하여 해결할 수 있음. 기존의 경우 EX stage 이후 그 값을 IF로 forwarding해야 하므로 2 cycle 동안 stall해야 함. pipeline이 길어지면 branch 연산의 결과를 빨리 결정할 수 없어 오랫동안 stall해야 하므로 stall penalty가 너무 커짐.

compiler scheduling으로 분기와 상관없는 명령어들을 우선 수행하도록 하여(delayed branch) 해결할 수 있음.

대부분의 시스템은 branch prediction을 주로 사용함.

2.6.2. branch prediction

Definition 81 *branch prediction은 branch 여부에 대해 예측하는 것임. branch prediction에는 static prediction과 dynamic prediction이 있음.*

branch 결과의 예측이 틀리면, fetch되어 수행한 명령어들에 대한 register 값을 삭제함으로써 동작을 취소하는 flush가 수행되어야 함. 수행한 결과를 삭제한 자리는 nop으로 취급됨.

분기 예측에 있어 분기가 이뤄지는 경우를 taken, 이뤄지지 않는 경우를 not taken이라고 함.

1. 정적 예측(static prediction)
: 전형적인 분기 경향을 정적으로 예측하는 것을 말함.
always taken 또는 always not taken으로 예측함.
특히 반복문 등에서는 대부분의 경우 (앞 또는 뒤로) 분기하고, 종료 조건에 걸릴 때에만 분기하지 않음.

pipeline이 길어질수록 예측 실패 시 penalty가 커지므로, 동적 예측을 주로 사용함.

2. 동적 예측(dynamic prediction)
: 실제 경향을 측정하여 동적으로 예측하는 것을 말함.
최근 수행된 branch 명령어의 taken history를 기록해두고 다음 branch 명령어가 경향성을 따라갈 것이라고 예측함.

이때 1-bit로 예측할 수도 있고, 2-bit로 예측할 수도 있음.

branch prediction은 주가 예측과 같이 성공하면 성능상 이득이고 실패하면 성능상 손해임.

branch prediction 외에도 branch에 대한 성능 향상을 위해 branch만을 위한 unit을 추가하여 연산을 한 stage만큼 당기고, 분기 target 주소를 저장하여 사용함.

사실 어플리케이션이 워낙 다양하다 보니 예측은 랜덤하게 예측하는 것보다 성능이 그렇게 개선되지는 않음. 한정된 어플리케이션을 사용하는 환경에서는 성능 상승 폭이 비교적 큼.

2.6.3. predictor

Definition 82 *BPB(Branch Prediction Buffer)/BHT(Branch History Table)는 dynamic branch prediction에서 taken history를 저장하는 unit을 말함. 이를 예측기(predictor)라고도 함.*

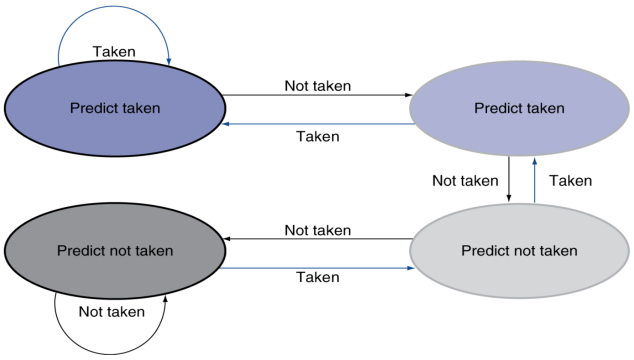
branch 명령어가 등장하면 BPB에서 해당 명령어의 경향성을 확인해 분기하거나 분기하지 않는데, 틀릴 경우 잘못 수행한 부분을 전부 flush하고 BPB의 값을 수정함.

BPB에는 최근에 발생한 각 분기 명령어의 주소를 index로 하여 taken 또는 not taken 여부를 비트 단위로 저장하는데, 이때의 비트 크기에 따라 종류를 나눌 수 있음. 즉, pc값을 인덱스로 값을 확인하여 해당 명령어가 분기할 것인지 예측함.

1. 1-bit predictor
1번의 taken 또는 not taken에 대한 history만을 저장함.
반복문의 경우 한 번은 반드시 잘못 예측할 수밖에 없음.
다중 반복문의 경우 내부의 반복문에 대해서 계속해서 잘못 예측하게 됨.

2. 2-bit predictor
아래의 다이어그램과 같이 한 번의 taken 또는 not taken으로 경향성을 바로 수정하지 않고, 특정 값이 연속적으로 들어온 경우에 값을 수정함.
경향성을 더 잘 반영할 수 있음.

처음에는 1비트를 가지고 예측을 했지만, 예측 실패가 필연적으로 발생하므로 현재는 주로 2비트를 사용함.

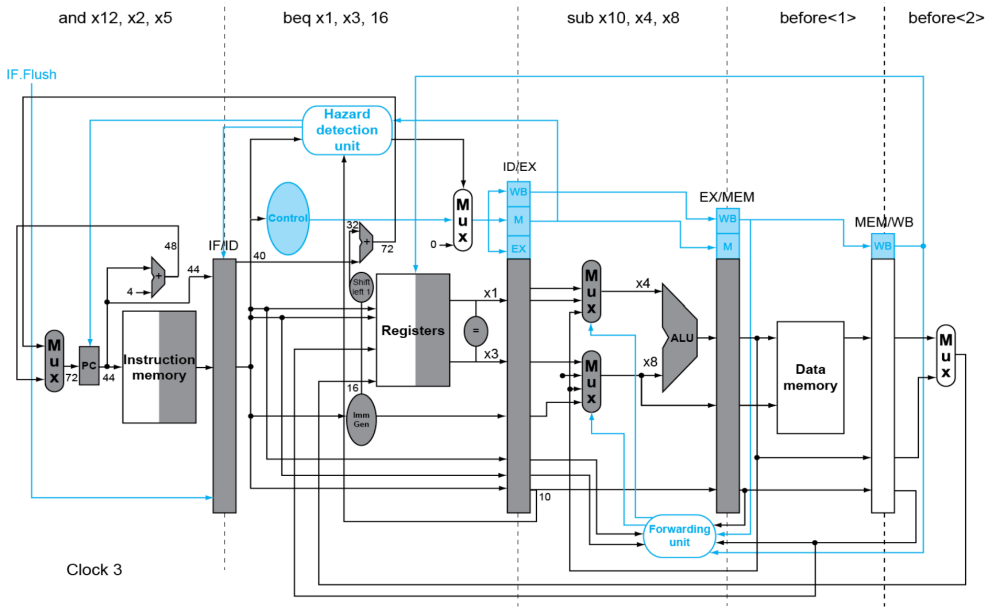


2.6.4. branch 연산을 위한 추가 unit

Definition 83 *branch 연산에 대해서는 주소 연산을 가능한 빨리 수행하여 그 결과를 사용하는 것이 유리함. beq 명령어를 위해 pipeline에 Target address adder(주소 가산기), Register comparator(레지스터 값 비교)를 추가하여 ID stage에서 연산을 수행할 수 있음.*

기존의 경우 stall로 해결하려면 2 cycle동안 stall해야 했지만, ID stage로 연산을 옮긴 경우 1 cycle만으로 해결할 수 있음.

branch 연산을 위한 추가적인 unit들이 추가된 pipeline은 아래와 같음.



참고로, 40: beq x1, x3, 16이라고 해보자. 16(imm12)에 해당하는 값은 0번째 비트가 생략된 것이므로 « 1 연산이 수행된 후 pc에 더해짐. 즉, x1과 x3가 같은 경우 72로 분기함.

2.6.5. branch 연산에 forwarding이 필요한 경우

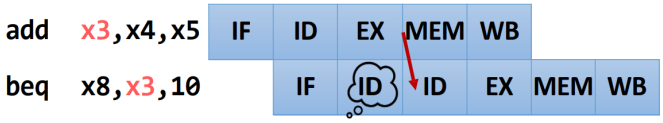
Definition 84 *직전에 수행된 명령어에 의해 branch 연산에 사용되는 rs1, rs2에 forwarding이 필요한 경우, 직전에 수행된 명령어의 종류에 따라 필요한 stall 수가 달라짐.*

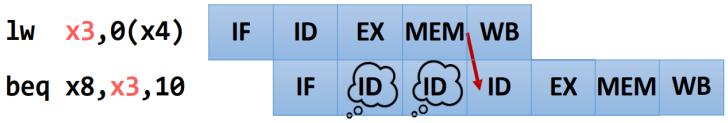
forwarding이 존재하지 않았다면 branch 연산을 위한 추가 unit으로 인해 stall해야 하는 cycle 수가 줄어들지만, 이는 EX의 연산을 ID로 미리 당겨서 하는 것이므로 forwarding이 존재한다면 stall 해야 하는 cycle이 발생하게 됨.

1. R-type 명령어
branch 연산을 위한 추가 unit을 사용하지 않은 경우 stall이 필요하지 않음.
branch 연산을 위한 추가 unit을 사용한 경우 1 cycle stall이 필요함.

2. load 명령어
branch 연산을 위한 추가 unit을 사용하지 않은 경우 1 cycle stall이 필요함.
branch 연산을 위한 추가 unit을 사용한 경우 2 cycle stall이 필요함.

시간선을 그리고 명령어 별로 stage를 나타내 보면 몇 번 stall해야 하는지 쉽게 확인할 수 있음.





2.6.6. BTB

Definition 85 *BTB(Branch Target Buffer)는 분기 target 주소를 저장하는 cache임.*

BTB에는 각 branch 명령어의 주소(해당 순간의 pc값)를 index로 하여 명령어에 대한 분기 target 주소를 저장함.

BTB에서 hit이 되었고, predictor에 의해 taken으로 예상되는 경우 명령어에 대해 연산을 수행하지 않고 BTB의 값을 그대로 가져와 분기함.

원래 같으면 분기할 것이라고 예측되는 경우에도

반복문 등에서의 연속적인 분기 시에 명령어 처리 속도를 빠르게 할 수 있음.

2.6.7. BPB/BTB의 동작 방식

Definition 86 *1. 동작 방식*
BPB/BTB는 IF stage에 존재하여 모두 pc값을 인덱스로 작업을 수행함.

BPB에 의해 taken으로 예측되는 경우 BTB에서 pc값으로 분기 주소를 찾아 분기함. 이때 BTB에 pc값에 해당하는 주소가 없다면 해당 stage에서 주소 계산을 하지 못하므로 stall이 발생할 수 있음. 물론 실제로 분기하는 경우에만 fetch한 것을 nop로 처리하고, 분기하지 않는 경우에는 nop 처리할 필요가 없음.

BTB에 pc값에 해당되는 주소가 있다면 바로 pc로 전달하여 stall 없이 분기할 수 있음. 물론 이후의 stage에서 실제로 계산했을 때 분기하지 않아야 했다면 분기한 위치의 명령어들을 전부 nop 처리해야 함.

BPB에 의해 not taken으로 예측되었는데 이후의 stage에서 실제로 계산했을 때 분기해야 했다면 수행한 명령어들을 nop 처리해야 함.

2. 성능
5-stage pipelined processor 에, IF stage 에 BTB/Predictor 를 구현했고, ID stage 에서 Branch comparison 이 계산된다면 최악의 경우 stall은 1번까지만 발생함.

물론 forwarding이 필요한 경우 등에서는 추가적인 stall이 필요함.

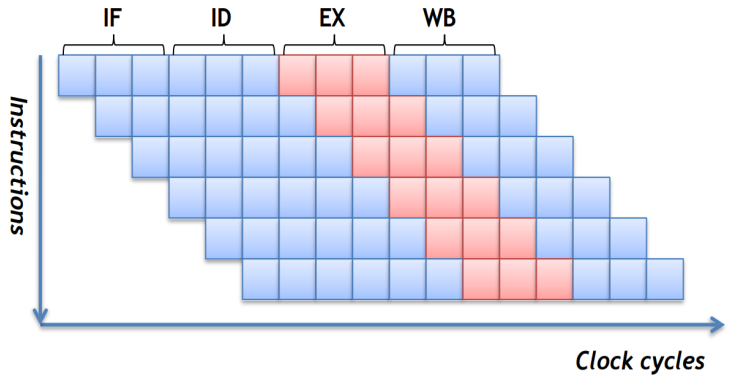
3. advanced processor

3.1. advanced processor

3.1.1. superpipelined

Definition 87 *superpipelined는 기존에 등장한 pipeline의 stage를 더 작게 나눈 것임.*

pipeline에서는 block 당 하나의 cycle을 사용하므로 clock speed가 훨씬 빨라짐.

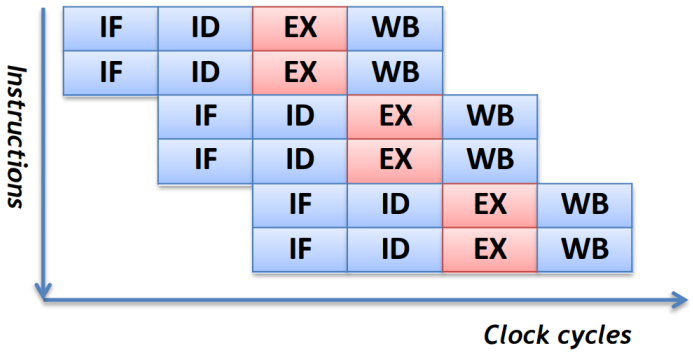


3.1.2. multiple-issue

Definition 88 *multiple-issue는 하나의 코어에서 동일한 pipeline 여러 개가 동시에 작동하여 여러 명령어를 병렬적으로 처리하는 것을 말함.*

물론 이때 더 많은 register와 unit들이 필요하고, register file에서의 추가 port 등이 필요함.

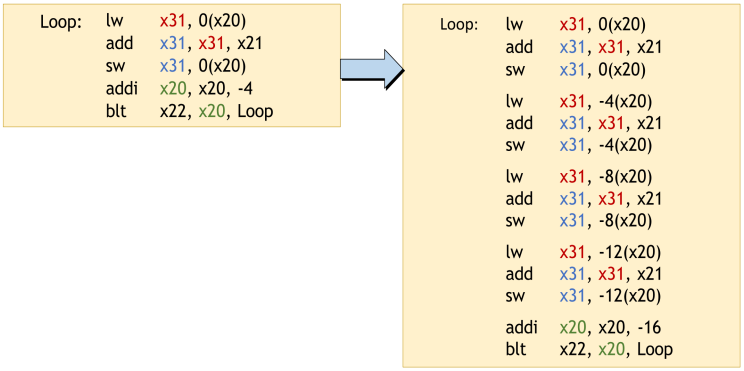
RISC-V에 적용해 보면 하나의 pipeline에서는 ALU/branch 명령어를, 다른 pipeline에서는 load/store 명령어를 수행하도록 구성할 수 있음. 특정 시점에 한 pipeline만 동작하고 있다면 동작하지 않는 pipeline은 nop로 처리함.



3.1.3. loop unrolling

Definition 89 *loop unrolling은 반복문(loop)에 대해 내부 명령어가 여러 번 수행되는 것을 취급하여 분기 횟수를 줄이는 것을 말함.*

성능은 좋아지지만 더 많은 register를 사용하고 저장해야 하는 코드의 양이 늘어나기 때문에, 임베디드 같은 적은 자원의 시스템에서는 잘 사용하지 않음.



Part V

메모리

1. memory

1.1. memory

1.1.1. memeory

Definition 90 *memory는 데이터를 저장하고 access하는 component를 말함.*

memory에 access하여 데이터를 가져오는 것을 참조(reference)한다고 하기도 함.

1.1.2. random access

Definition 91 *저장공간 내에서 데이터가 어느 위치에 존재하는지에 상관없이 일정한 access time을 보장하기 위해 random access를 함. access time에 대한 예측이 가능한 것이 특정 데이터에 대해 성능을 약간 개선하는 것보다 성능과 설계 면에서 더 유리함. 실제로 평균적인 경우 non-random access는 random access에 비해 현저히 느린 access time을 가짐.*

random access를 사용하는 memory로는 DRAM, SRAM, Flash memory가 있음.

non-random access를 사용하는 memory는 접근 순서의 필요성에 따라 순차적(sequential)/비순차적(non-sequential)으로 나뉨. 순차적 non-random access에는 magnetic tape가 있고, 비순차적 non-random access에는 HDD(hard disk)가 있음.

현재 non-random access는 잘 사용하지 않음.

참고로 HDD는 LP판을 겹쳐놓고 읽는 것처럼 동작함.

1.1.3. RAM

Definition 92 *RAM(Random Access Memory)는 random access를 사용하는 memory로, DRAM, SRAM, Flash memory(SSD)가 있음.*

memory에서 density란 면적 대비 데이터 용량을 말함.

1. DRAM(Dynamic Random Access Memory)
: 데이터가 주기적으로 refresh되어야 하는(dynamic) RAM.
row 단위(4KB 정도)로 사용됨.
높은 density, 낮은 전력소모, 낮은 가격, 느림.
전원이 끊기면 데이터가 소실됨(volatile memory).

비트당 1개의 트랜지스터를 사용하여 해당 비트의 전하량을 확인함.

계속해서 refresh해야 하는 특성을 leaky bucket이라고 하기도 함. capacitor(양동이)에 전하(물)가 차면 1, 비어있으면 0으로 취급하는데 이 전하가 계속 새는 것(64ms 정도). 특히 온도가 높아지면 전자의 운동이 활발해져 더 빠르게 샘(8∼32ms 정도). 또한 해당 capacitor가 차 있는지를 검사할 때 전하를 비워보기 때문에, read 후 다시 채워줘야 함.

main memory는 주로 DRAM으로 되어 있음.

2. SRAM(Static Random Access Memory)
: refresh하지 않아도 데이터가 유지되는(static) RAM.
낮은 density, 높은 전력소모, 높은 가격, 빠름(DRAM의 2∼10배).
전원이 끊기면 데이터가 소실됨(volatile memory).

DRAM은 비트당 1개의 트랜지스터를 사용하지만 SRAM은 비트당 4∼6개의 트랜지스터를 사용하기 때문에 density 가 낮고, 전력을 많이 쓰고, 비싼 것.

register는 주로 SRAM으로 되어 있음.

3. Flash memory(SSD)
: 영구적인 데이터 저장에 사용되는 non-volatile memory.
block 단위로 데이터가 작성됨.
전원이 끊겨도 데이터가 소실되지 않음(non-volatile memory). 물론 SSD도 전원을 인가하지 않고 몇 년간 방치하면 데이터가 소실됨.

SRAM은 캐시메모리로 주로 사용하고, DRAM은 프로그램 메모리에 주로 사용함. 물론 돈만 많다면 전부 SRAM을 사용하는 것이 성능이 좋음.

1.1.4. non-volatile memory

Definition 93 *비휘발성 메모리(non-volatile memory)는 전원이 연결되어 있지 않아도 데이터가 유지되는 memory로, ROM, PROM, EPROM, EEPROM, Flash memory, Optane memory 등이 있음.*

1. ROM(Read-only memory)
: 공장에서 프로그래밍되고 이후 다시 수정할 수 없는 memroy.
펌웨어 프로그램(BIOS 등)같이 수정될 필요가 없는 데이터를 저장함.

2. PROM(Programmalbe memroy)
: 한 번만 프로그래밍이 가능한 memory.
주로 제조사 등에서 한 번 프로그래밍할 수 있도록 한 것.

3. EPROM(Erasable PROM)
: 내용을 지우고 다시 작성할 수 있는 PROM.

4. EEPROM(Electrically erasable PROM)
: UV, X-ray 등으로 내용을 지우고 다시 작성할 수 있는 PROM.

5. Flash memory
: 부분적으로(block 단위) 내용을 지우고 다시 작성할 수 있는 EEPROM.
NOR, NAND가 존재함.
SSD에 사용됨. SSD에는 NAND Flash가 사용됨.

6. Optane memory
: Intel에서 개발한 DRAM과 flash memory 사이의 non-volatile memory.
제작엔 성공했지만 비싸서 사라짐.

1.2. memory 성능 개선

1.2.1. memory 성능 개선

시간이 지남에 따라 memory(셀) 자체의 성능(access 속도) 증가는 프로세서에 비해 굉장히 더뎠고, memory 크기가 늘어나게 됨에 따라 병목현상이 일어남. 하지만 그렇다고 빠른 memory만을 사용하면 가격이 너무 비싸짐. 이에 따라 memory에 설계적인 측면에서의 개선이 이루어짐.

hybrid solution으로, memory의 hierarchy를 구성함. 자주 사용되는 데이터는 빠르고 비싼 memory(cache)에 저장하고, 그렇지 않은 데이터는 느리고 싼 memory에 저장함.

참고로, memory의 크기가 커질수록 처리해야 하는 주소가 많아져 속도가 느려짐.

1.2.2. locality

Definition 94 *시간/공간적으로 전체 주소 중 특정 부분이 주로 사용되는 것을 데이터/명령어의 지역성(locality)이라고 함.*

90/10 rule에 의하면 90%의 access는 전체 주소의 10%에서만 일어남.

locality는 아래와 같이 두 가지로 나뉨.

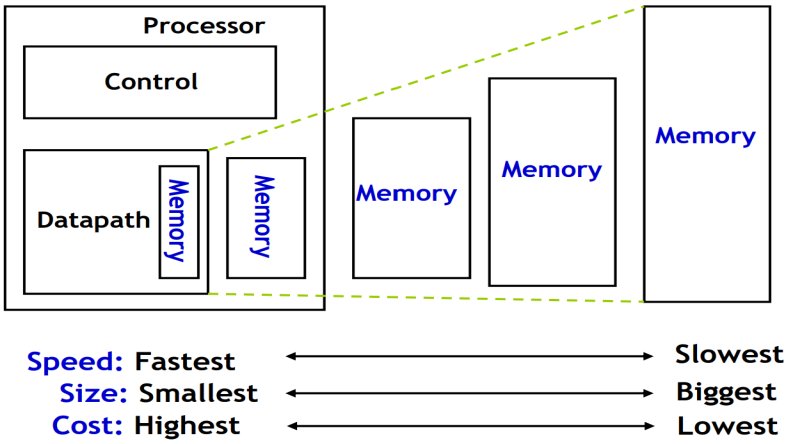
1. 시간적 지역성(Temporal locality)
: 어떤 데이터/명령어가 최근에 사용되었다면 다시 사용될 가능성이 높음.
ex. 반복문.

2. 공간적 지역성(Spatial locality)
: 어떤 데이터/명령어가 사용되었다면 그 주변 데이터가 사용될 가능성이 높음.
ex. 배열.

memory를 여러 개의 hierarchy로 분리하여 성능을 향상시킬 수 있는 것은 데이터의 locality 때문임.

1.2.3. hierarchy

Definition 95 *locality에 의해 memory의 hierarchy를 구성하여 빠른 memory를 주로 사용할 수 있음.*
크게 보면 레지스터-cache-DRAM의 구조로 생각할 수 있음.



1.2.4. hierarchy 관련 용어

Definition 96 *1. block*
서로 다른 hierarchy에서 주고받는 데이터의 단위를 block이라고 함.

cache끼리도 block 단위로 데이터를 주고받는데, upper level로 갈수록 block의 크기가 점점 작아짐. 이때 시스템에 따라 upper level과 block 크기가 같을 수는 있어도, upper level block 크기가 lower level block 크기보다 커지게 설계하지는 않음.

2. hit
어떤 계층에서 access했을 때 해당 데이터가 존재하면 hit라고 함.
hit rate(비율) = hit 횟수 / access 횟수
hit time은 해당 level의 cache에서 access하는 데 걸린 시간.

upper level의 hit rate이 높을수록 성능이 좋음.

3. miss
어떤 계층에서 access했을 때 해당 데이터가 존재하지 않으면 miss라고 함.
miss rate = 1 - hit rate
miss penalty = 다음 level의 access time + 해당 level에 가져온 데이터를 저장하는 데 걸리는 시간

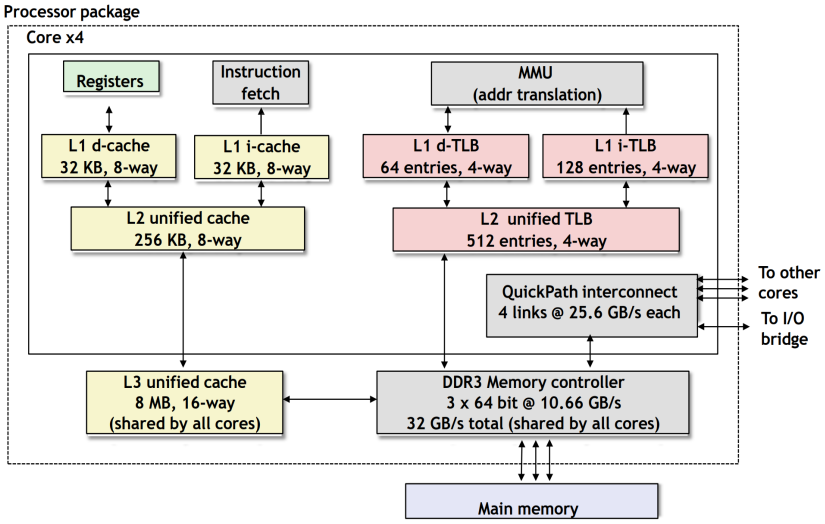
hit time에 비해 miss penalty가 압도적으로 큼.

어떤 level에서 miss가 났다면 다음 level에서 데이터를 다시 찾는데, 이 과정을 hit될 때까지 반복함.

1.3. memory 요약

1.3.1. hierarchy 구조

아래에서 설명할 memory의 구조를 종합하면 아래와 같음. 대부분의 cpu core는 이런 구조를 가지고 있음.
L1 cache와 TLB 모두 data/instruction 부분이 구분되어 있음.



1.3.2. hierarchy principles

Definition 97 *memory hierarchy의 각 level 모두에 적용할 수 있는 공통 원칙은 아래와 같음.*

1. 블록 할당(block placement)
block이 memory에 저장되는 것은 associativity에 따라 성능이 달라짐.
associativity를 늘리면 miss rate을 줄일 수 있지만 하드웨어 복잡도, 연산량(비용), 접근 시간을 증가시킴.

2. 블록 탐색(finding a block)
block 탐색 시에도 associativity에 따라 성능이 달라짐.
associativity만큼의 비교 연산을 수행해야 함. n-way인 경우 n번의 연산이 필요함.

3. replacement
memory 공간이 꽉 찼을 때 어떤 block을 삭제할 것인지는 주로 LRU를 따름.
또는 무작위로 삭제하기도 하는데, LRU와의 성능 차이가 그렇게 크지 않으면서 구현이 간단함.

4. write policy
memory에 내용을 작성하는 write policy로는 write-through, write-back 등이 있음.

1.3.3. write policy

Definition 98 *memory에 내용을 작성하는 write policy로는 write-through, write-back 등이 있음.*

일반적으로 upper level memory에는 lower level memory의 내용이 올라가고, lower level memory는 upper level memory의 데이터를 포함하고 있음. (대체로 이렇게 설계함.) 이에 따라 특정 memory의 block 내용이 수정된 경우 하위 level에 이를 언제 반영해 줄 것인지를 policy로 지정하는 것.

1. write-through
block의 데이터가 수정될 때마다 하위 level에 반영해주는 방식.

값을 미리 반영해뒀기에 block 교체는 간편하지만, 매번 쓰는 과정의 성능 향상을 위해 write buffer를 추가로 사용해야 함.

2. write-back
block이 삭제될 때 하위 level에 반영해주는 방식.

dirty bit 등의 사용이 필요함. block 내용에 수정이 발생했음을 dirty bit로 나타내고, replacement 시에 해당 block의 내용을 반영해주는 것.

write-back이 일반적으로 더 빠르기 때문에 VM 등에서는 write-back 방식만을 사용함.

1.3.4. architecture design에 따른 성능 변화

cache 크기가 증가하면, capacity miss는 줄어들지만 대체로 access time이 증가함.
associativity가 증가하면, conflict miss는 줄어들지만 대체로 access time이 증가함.
block size가 증가하면, compulsory miss는 줄어들지만 대체로 miss penalty와 miss rate이 증가함.

memory 접근이 프로세서의 연산 중 가장 느리기 때문에, memory 시스템의 설계가 프로세서의 성능에 있어 가장 중요한 부분 중 하나임.

2. cache

2.1. cache

2.1.1. cache

Definition 99 *캐시(cache)는 낮은 hierarchy(DRAM, Flash memory)의 Buffer로 사용되는 중간 단계의 저장 공간임. cache는 processor 내부에 존재함.*

1. level
cache는 높은 hierarchy에서 낮은 쪽으로 L1, L2, L3 등이 존재하는데, 프로세서에 가까울수록 upper level cache라 하고, 멀수록 lower level cache라 함. 대부분의 컴퓨터 시스템에서는 L1 L2/L1 L3의 다중 cache를 사용함.

L1은 접근하는데 1 4cycle, L2는 접근하는데에 30cycle까지 걸림.

2. locality
spatial locality를 이용하기 위해 연속적인 word들로 구성된 block을 cache에 저장함.
temporal locality를 이용하기 위해 최근에 접근한 데이터를 upper level cache에 저장해두고, 특정 level cache에서 데이터를 삭제해야 할 때에도 가장 과거에 접근한 데이터를 삭제함.

3. access time
hierarchy의 access time 계산 방법을 따름.

main memory 또한 storage의 cache로서 사용되는 것임.

2.1.2. multi-level cache

1. multi-level
cache는 L1, L2, L3 등 multi-level로 사용함. L1 cache는 특별히 primary cache라고도 함.

upper level cache일수록 더 작고, 빠르고, 더 작은 block size를 가지고, 더 낮은 associativity를 가짐.

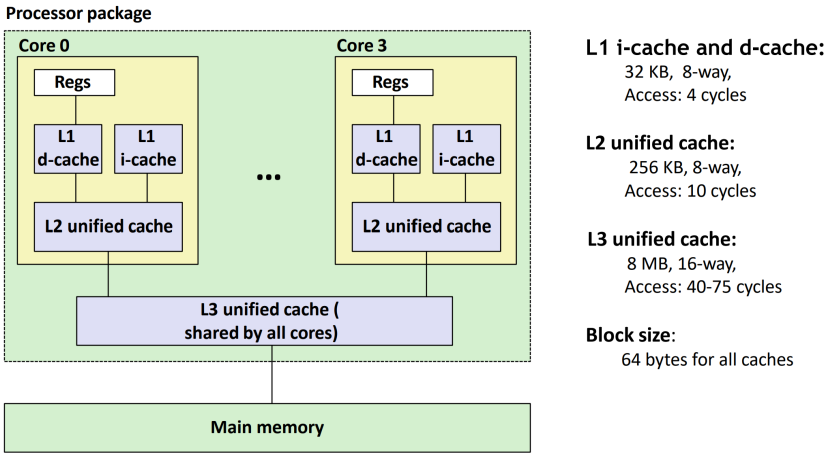
L2은 L1보다는 느리고 크지만, main memory보다는 빠르고 작기 때문에 miss penalty를 줄여 줄 수 있음. 특히 L3에 도달하는 access는 매우 적지만, L3 없이 매번 main memory에 직접 접근하게 된다면 miss penalty가 엄청나게 커짐.

대부분의 시스템에서는 L1, L2, L3만 사용하고, L4는 거의 사용되지 않음. 시스템의 크기가 작다면(임베디드 등) L1만, 또는 L1과 L2만 사용하기도 함.

하나의 cache만 사용하는 single-level cache의 크기보다 multi-level cache의 primary cache가 일반적으로 더 작음. 또한 single-level cache의 크기보다 L2의 크기가 더 큼.

2. level 별 주안점
대부분의 hit이 primary cache와 L2에서 발생하므로, 이 부분에서는 hit time 자체를 줄이는 것이 중요함.

3. 실제 구성 예시(intel core i7)
전에 등장한 내용처럼, cache는 내부적으로 data 부분(d-cache)과 instruction 부분(i-cache)이 분리되어 있음. 시스템에 따라 다를 수 있지만, L1에 대해서는 data/instruction이 분리되어 있고, L2부터는 이 둘을 합친 unified cache임. L3는 모든 코어에서 공유 사용하는 부분임.



2.1.3. cache miss

Definition 100 *cache의 miss에는 아래와 같은 것들이 있음.*

1. compulsory miss/Cold start : 처음 프로그램 실행 시에 강제적으로(compulsory) 발생할 수밖에 없는 miss.
block size를 늘려서 최초에 가져오는 데이터 양을 많게 하여 줄일 수 있음. 최초 실행 시에는 우선 데이터를 많이 가져오는 것.

2. conflict miss
: 매핑된 특정 entry에 다른 데이터가 있어 발생하는 miss.
cache 크기 또는 associativity를 늘려 줄일 수 있음.

conflict miss로 인해 기존의 데이터를 삭제해야 되는 상황을 thrashing이라고 함.

3. capacity miss
: memory 용량의 한계로 인해 발생하는 miss.
cache 크기를 늘려 줄일 수 있음.

2.2. cache arrangement

2.2.1. cache arrangement

Definition 101 *cache arrangement는 cache가 내부적으로 데이터를 정렬/저장하는 방법을 말하는데, direct mapped, fully associative, N-way set associative 등이 있음.*

2.3. direct mapped

2.3.1. direct mapped

Definition 102 *direct mapped는 각 memory block이 cache 내부에 존재하는 하나의 single cache block에 directly mapping되는 방식임.*

2.3.2. 저장 위치의 결정

Definition 103 *direct mapped에서 memory block을 cache block으로 올려 저장할 때는 다음과 같이 주소 비트를 사용함.*

각 memory block이 어떤 cache block에 매핑되는지는 해당 memory block의 주소를 cache entry 개수로 modular 연산을 하여 정함. 즉, 각 cache block에는 저장될 수 있는 memory block들의 집합이 정해져 있고, 이를 modular 연산으로 정하는 것임.

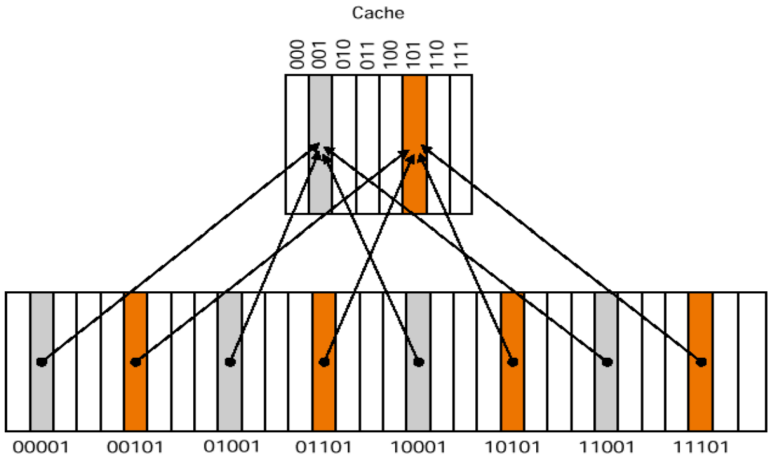
이때 나머지는 cache entry 개수(나누는 수)보다 항상 작으므로 주소 비트 중 하위 몇 개의 비트(LSB)만 가지고 연산을 수행할 수 있음. 이때의 하위 비트들을 index라고 함. 이때 cache entry로 나누는 것은 결과적으로 해당 주소의 index 비트를 그대로 가져오는 것과 같음.

index 비트의 크기는 cache entry의 크기에 의한 것임. cache entry 전부를 구분할 수 있도록 하는 만큼의 비트가 필요한 것.

특정 cache block에 들어갈 수 있는 주소들의 집합이 존재하고, 해당 집합에 속하는 memory block들은 한 번에 하나씩만 저장될 수 있음.

index 지정에 대한 예시로, cache entry가 0~7이라면 index는 3bit 크기를 가짐. 또한 11010111이면 하위 3개의 비트 111이 index가 되고 111에 해당하는 cache block으로 매핑됨.

direct mapped에서 각 memory block은 아래와 같이 주소 값에 따라 cache block에 매핑됨.



이렇게 데이터를 저장해 둠으로써 temporal locality를 활용함.

2.3.3. address의 분해

Definition 104 *direct mapped에서는 memory address를 tag, index, block offset으로 분해하여 cache에 데이터를 저장하거나 cache에서 데이터를 찾음.*

1. tag
index에 의해서 cache entry가 결정되는데, 해당 cache entry에 들어올 수 있는 데이터가 여러 개일 수 있으므로, 해당 데이터의 주소 중 일부를 tag로 사용하여 데이터를 구분함.

hit 여부를 판정하고 데이터를 가져다 쓰려면, 특정 cache block이 어떤 memory block의 데이터를 저장하고 있는지를 판단할 수 있어야 함.

tag는 주소 값 전체로 할 수도 있지만, 동일한 index 값을 가진다면 index를 제외한 부분만으로 데이터를 판별할 수 있으므로 상위 비트들을 tag로 할 수 있음.

2. offset(byte offset)
메모리의 접근은 block 단위로만 이루어짐. 즉, block 크기의 배수 만큼의 주소만을 사용함. 이에 따라 address에서는 block 크기만큼의 부분에 항상 0만이 들어가 있으므로 이를 제외하고 index와 tag로 사용하는데, 이 부분을 offset(byte offset)이라고 함.

(아마 메모리의 접근은 word 단위로만 이루어지고, 주소 또한 word의 배수만이 존재한다고 하는 것이 정확할 것이다. 교수님이 헷갈리신 듯? 그렇지 않으면 multi-word block에서 block offset으로 각 word를 구분할 수가 없다.)

direct mapped에서 각 주소 비트는 아래와 같이 tag, index, offset으로 구분되어 사용되고, 각각의 크기는 시스템마다 다를 수 있음.

2.3.4. direct mapped의 구현

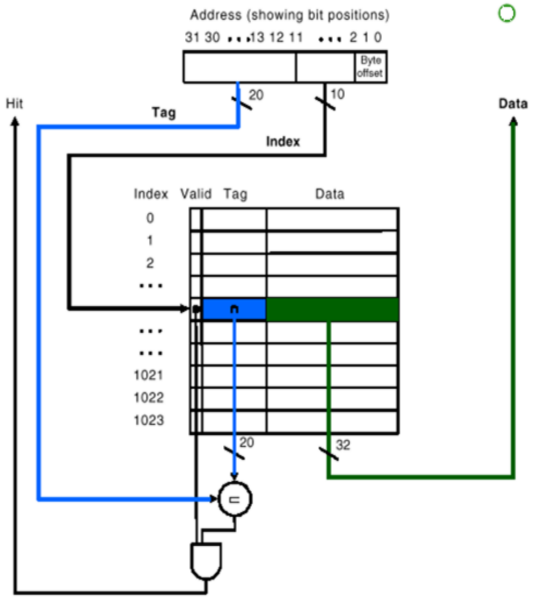
Definition 105 *direct mapped cache에는 데이터를 저장하는 부분 외에도, 저장된 데이터를 구분하기 위한 tag와 해당 cache block에 데이터가 들어 있는지를 나타내는 valid bit가 존재함. 이때 valid bit의 초깃값은 0으로, 데이터가 존재하면 1, 존재하지 않으면 0으로 지정함.*

정리하면, 주소 비트에서 offset을 제외하고 필요한 만큼을 어떤 cache block에 매핑되는지를 판단하는 index로 사용함. 그리고 남은 비트를 해당 cache block에서 데이터를 구분하는 tag로 사용함. cache에는 valid bit, tag, 실제 데이터가 저장되는 것.

참고로 cache의 전체 크기는 valid bit, tag를 제외한, 저장될 수 있는 실제 데이터의 크기를 말함. 예를 들어, 4KB cache라면 실제로 cache가 정상적으로 동작하기 위해 가져야 하는 memory의 크기는 4KB 보다 큼.

direct mapped의 동작 과정을 정리하면 아래와 같음. 그 아래의 그림은 이 과정에 대한 회로임. valid bit가 연결된 회로는 AND gate이고, tag가 연결된 회로는 두 값이 같은지를 검사하는 회로임.

1. access 시에 주소가 입력되면 index 부분을 확인하여 특정 cache block으로 이동함.
2-1. valid bit가 1이면 tag를 가져와 비교함.
2-2. valid bit가 0이면 hit 신호를 0으로 보내고 다음 level로 넘어감.
3-1. tag가 같으면 hit 신호를 1로 보내고 data를 전송함.
3-2. tag가 다르면 hit 신호를 0으로 보내고 다음 level로 넘어감.



2.3.5. multi-word block

Definition 106 *여러 개의 word만큼의 block 크기를 사용하여(block size를 늘려서) spatial locality를 활용할 수 있음.*

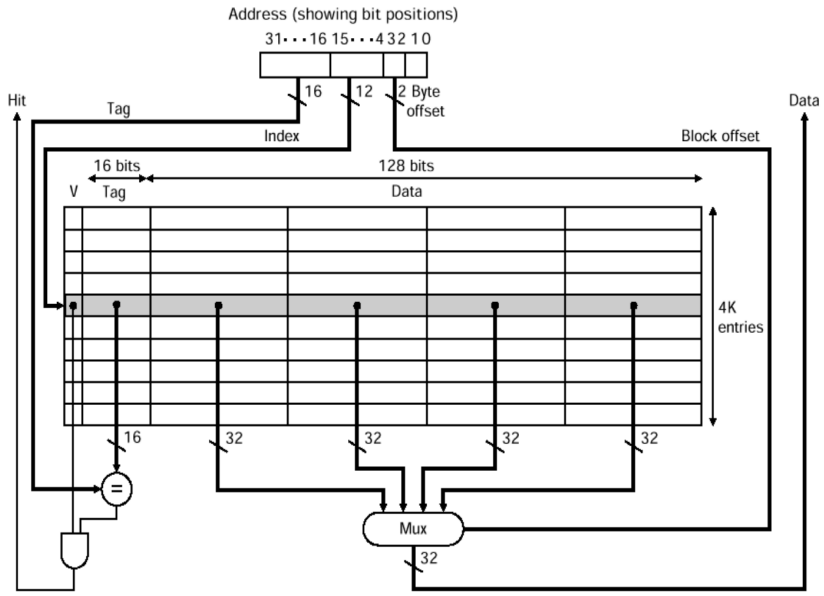
여러 개의 word로 block이 구성되어 있다면 cache에 저장된 해당 block에서 특정 word의 데이터를 사용할 수 있어야 함. block 내에서 각 word를 구분하기 위해 추가로 block offset을 사용함. block offset으로는 주소에서 byte offset 다음 하위 비트들을 사용하며, block 내 word를 구분할 수 있을 만큼의 bit를 가짐.

byte offset에는 항상 0만이 들어가 있지만, block offset에는 그렇지 않음. 즉, 하나의 word로 구성된

block과 동일하게 사용하는데 내부적으로 하나의 block이 여러 개의 붙어 있는 word를 가지고 있어 spatial locality를 사용하게 되는 것임.

저장 시에는 word가 하나일 때와 동일한 방식으로 저장하면 됨.

reference 시에도 word가 하나일 때와 유사한데, hit되고 데이터 접근 시에 block offset을 신호로 mux가 동작하여 적절한 word를 선택한다는 점에 차이가 있음.

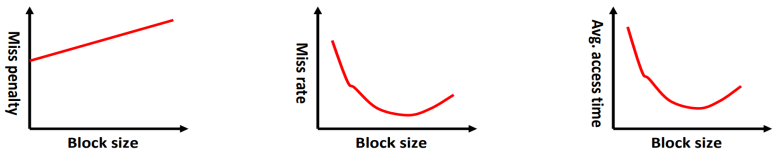


2.3.6. 적절한 block size의 지정

block size를 늘리는 것은 miss rate를 줄여줄 수 있지만, cache의 크기가 동일한 경우 block size가 너무 커지면 그만큼 실제로는 사용되지 않아 낭비되는 공간이 많아지기 때문에 오히려 miss rate가 증가하게 됨. 또한 데이터를 올리고 삭제하는 데에 걸리는 작업량이 늘어나 miss penalty가 증가함.

너무 큰 block은 성능에 악영향을 미친다고 해서 pollution이라는 표현을 사용하기도 함.

L1, L2 cache에서는 block size가 64byte정도가 optimal하고, L3 cache같이 더 lower한 memory에서는 128/256btye를 사용하기도 함.



2.3.7. direct mapped의 문제점

1. conflict miss가 자주 발생할 수 있음.

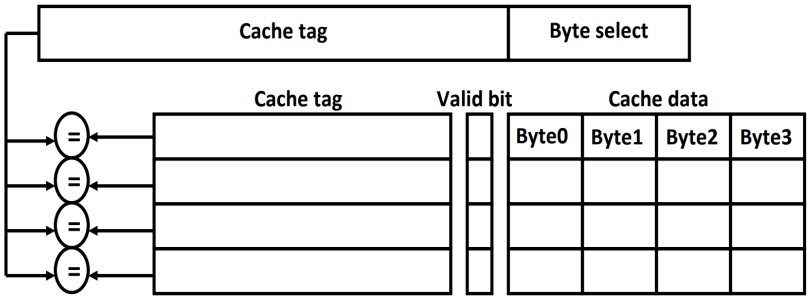
2. conflict miss로 인해 데이터를 교체한 것이 또 다른 conflict miss를 일으킬 수 있음.

2.4. fully associative

2.4.1. fully associative

Definition 107 *fully associative는 각 cache entry에 어떤 주소 값을 가지는 데이터라도 저장될 수 있도록 하는 방식임.*

이 경우 conflict miss가 발생하지 않고, capacity miss만 발생하게 됨. 다만, 특정 데이터를 찾을 때 cache 내에 존재하는 모든 entry의 tag와 비교 연산을 수행해야 함.



2.5. N-way set associative

2.5.1. N-way set associative

Definition 108 *N-way set associative는 데이터가 cache 내부에서 속하는 특정 set 안에서는 어떤 위치에도 저장될 수 있도록 하는 방식임.*

1. N
N-way set associative에서는 N개의 entry가 각 index에 할당되는데, 이는 N개의 direct mapped cache가 병렬적으로 동작하는 것과 같음. 즉, 하나의 index에 대해서 서로다른 tag를 가지는 N개의 데이터를 저장할 수 있고, 이에 따라 서로 conflict인 N개의 데이터를 conflict 없이 저장할 수 있음.

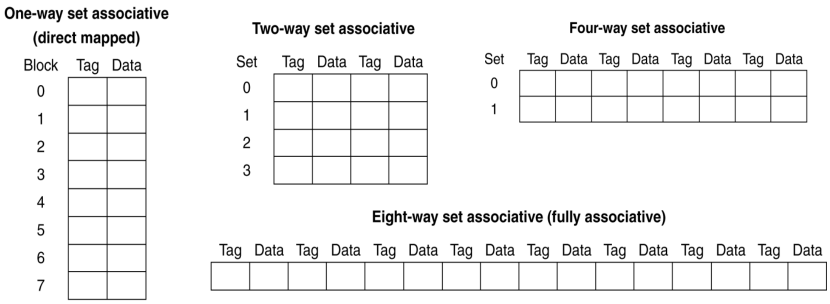
이때 하나의 index에 대한 entry의 집합을 set이라고 함.

하나의 set에 대해서 특정 데이터를 찾으려면 각 tag에 대한 N번의 비교 연산을 수행해야 함.

2. 데이터의 삭제
conflict가 발생하면 해당 set의 데이터들 중 어떤 데이터를 삭제할 것인지 선택해야 하는데, 이는 나름의 replacement policy를 따름.

direct mapped와 fully associative의 절충안으로, direct mapped의 단순한 구현과 fully associative의 conflict miss 회피라는 특성을 차용함. direct mapped에서 관찰한 결과 특정 working set(작업의 집합)에서 2~4개 주소들의 conflict miss가 주로 발생했는데, 이에 따라 하나의 매핑 당 2~4개의 conflict miss 정도는 피할 수 있도록 구현한 것.

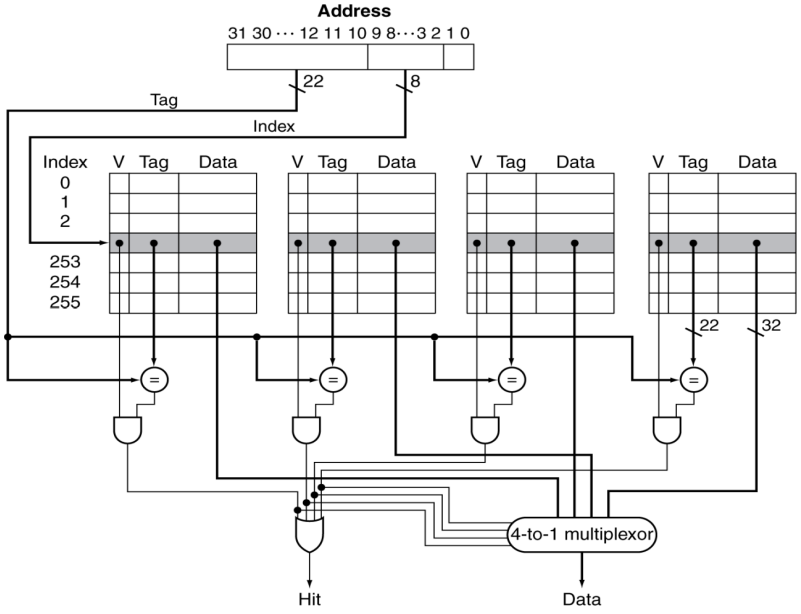
direct mapped의 multi-word block과의 차이점은, block size만을 늘린 것은 해당 데이터 주위의 데이터를 함께 저장해 spatial locality를 활용한 것이고, N-way set associative에서는 서로 떨어져 있고 index 값만이 같은 데이터들의 conflict를 해결했다는 것임. 아마 실제 구현에서는 N-way set associative의 각 병렬적 direct mapped cache가 multi-word block을 사용하도록 할 것임.



2.5.2. N-way set associative의 구현

N-way set associative에서 데이터를 저장하는 것은 direct mapped와 유사한데, 하나의 index에 대한 공간이 여러 개이므로 잘 선택해서 넣어주면 됨.

reference 시에도 index를 비교하고 valid bit를 검사하는 것은 direct mapped와 동일하지만 이후 tag를 비교하는 연산을 각 병렬적 direct mapped cache에 대해 모두 수행한다는 점에 차이가 있음. 어떤 공간에서 hit가 났는지에 따라 해당 데이터를 전송함.



2.5.3. associativity

Definition 109 *N-way set associative에서의 N이 커지면 associativity가 커진다고 하고, N이 작아지면 associativity가 작아진다고 함.*

associativity가 증가하면 miss rate는 줄어들지만(conflict miss 감소), 비교 연산을 수행해야 하므로 연산량이 늘어남. 또한 어느 정도 associativity가 증가하면 miss rate 감소 폭이 작아짐. 시스템에 적절한 수준의 associativity를 설정해야 함.

associativity을 적절히 증가시키면 비교 연산량이 많아지므로 hit time이 늘어나지만, miss rate을 줄일 수 있기 때문에 결과적으로 average memory access time은 줄일 수 있음.

일반적으로 L1은 2∼4-way, L3는 8∼16-way 정도를 사용함. N이 너무 커지면 매번 수행해야 하는 비교 연산이 많아지므로, fully assiciative와 동일한 문제를 가지게 됨.

associativiy가 높다고 항상 miss rate이 더 낮은 것은 아님. cache 크기에 따라 2-way의 성능이 4-way보다 좋을 수도 있음. 물론 일반적으로는 4-way가 2-way보다 miss rate이 낮음.

2.5.4. replacement policy

Definition 110 *N-way set associative의 replacement policy는 set에서 데이터를 삭제해야 할 때 어떤 것을 우선적으로 삭제할지에 대한 policy임.*

LRU(Least Recently Used) replacement policy는 가장 과거에 사용되었던 데이터를 우선적으로 삭제하는 정책(policy)/알고리즘임.

새로운 데이터를 저장해야 할 때 공간이 비어 있는 경우 그냥 넣으면 되지만, 빈 공간이 없는 경우 LRU replacement policy를 따름.

direct mapped cache에서는 하나의 set에 하나의 데이터만이 존재하므로 어떤 데이터를 선택하여 삭제할지에 대한 판단이 필요하지 않음.

temporal locality를 활용하기 위해 가장 과거에 사용되었던 데이터를 삭제하는 것.

2.6. cache performance

여기서는 miss penalty와 그에 따른 CPI로 성능을 살펴봄.

2.6.1. cache performance

Definition 111 *cpu가 어떤 프로그램을 수행하는 시간은 아래의 구성 요소를 가짐. 즉, 명령어가 실제로 수행되는 시간과 miss가 난 경우의 평균적인 penalty를 더한 것임.*

1. Program execution time : 명령어 수행, hit time 등 포함.
2. Miss penalty : cache miss로 인한 stall time.

miss가 나는 경우의 stall cycle(Memory stall cycle)을 구하는 수식은 아래와 같음. 즉, 메모리에 접근하는 명령어의 비율과 그 중 miss가 발생하는 비율, miss 시의 penalty를 곱한 것임.

```
Rate of memory acess i = Memory access instructions / total instructions
Memory stall cycles = Rate of memory acess i * Miss rate * Miss penalty
```

어떤 명령어의 수행에서는 I-cache에서 명령어를 가져오고, memory 접근 명령어의 경우 D-cache에 접근해야 함. 그러므로 평균 CPI을 구할 때는 아래와 같이 기존의 계산으로 구한 CPI(base CPI)에 miss로 인해 발생하는 평균 cycle을 더해줘야 함. 이때 I-cache에는 모든 명령어가 접근하지만, D-cache에는 특정 명령어들만 접근하므로 해당 비율을 반영해 줘야 함.

```
Actual CPI = Base CPI + I-cache miss cycles + D-cache miss cycles
```

아래의 상황은 예시임.

- Given
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- Miss cycles per instruction
  - I-cache: 0.02 x 100 = 2
  - D-cache: 0.36 x 0.04 x 100 = 1.44
- Actual CPI = 2 + 2 + 1.44 = 5.44
  - Ideal CPU is 5.44/2 = 2.72 times faster

당연하게도, CPI의 계산은 cycle 수로 해야 하기 때문에 시간이 주어졌다면 몇 cycle인지 계산해 줘야 함.

cpu logic의 성능이 증가하며 miss penalty를 줄이는 것이 더 중요해졌음. 또한 CPI가 줄어들면서 stall이 더 많은 cycle을 멈춰야 하게 되었음. 시스템 성능 평가 시에 cache의 동작이 더 큰 영향력을 가지게 됨.

2.6.2. access time

Definition 112 *AMAT(Average Memory Access Time)는 특정 level memory의 평균 access time인데, 이는 아래의 수식으로 구할 수 있음. hit되었을 때의 시간과 miss가 났을 때의 시간을 더해 준 것.*

```
AMAT = Hit time + Miss penalty * Miss rate
```

miss penalty는 다음 level로의 access time과 데이터 저장 시간을 더한 것이므로 아래의 수식으로 구할 수 있음. miss penalty는 다음 level의 access time을 사용하므로, multi-level hierachy의 경우 miss penalty와 access time은 재귀적으로 정의됨.

```
Miss penalty = Next level access time + Data saving time
```

예를 들어, L1 (average) Access time = L1 hit time + L1 miss penalty * L1 miss rate이고, L1 miss penalty = L2 access time + L1 저장 시간임.

당연하게도, multi-level cache의 cpi(성능) 계산 시에 miss penalty이나 miss rate가 global하게 제공되었다면 재귀적으로 계산해서는 안 됨.

3. virtual memory

3.1. virtual memory

3.1.1. virtual addressing

Definition 113 *virtual addressing은 cpu에서 하드웨어 상의 실제 주소(physical address)를 사용하는 대신 가상의 address 체계를 사용할 수 있도록 하는 것을 말함. 이때 virtual addressing에 의해 사용할 수 있는 가상의 address 체계의 가상 memory를 VM(virtual memory)이라고 하고 이때의 address를 VA(virtual address). 반대로 실제 memory의 주소는 PM(physical memory)라고 하고 이때의 address를 PA(physical address).*

cpu에서는 virtual memory를 사용하고, MMU라는 unit이 이 VA를 PA로 변환하여 PA에 실제 값이 load/store되도록 함.

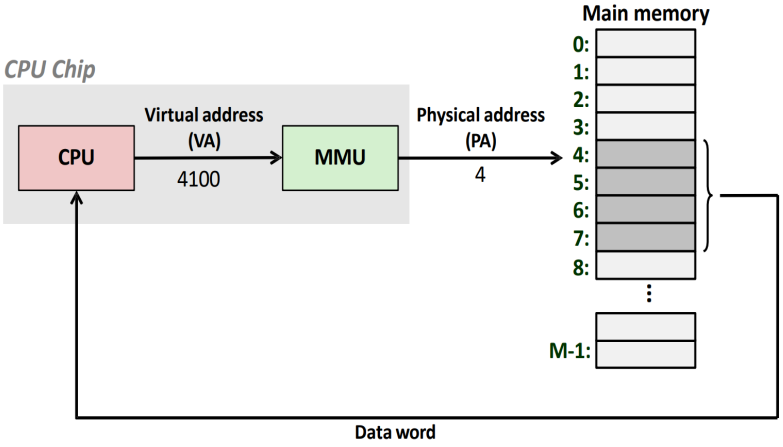
각 프로세스에서는 main memory 전체를 혼자서 사용하는 것처럼 동작하도록 되어 있음.

virtual addressing은 아래와 같은 이점들을 제공함.

1. PA를 고려하지 않고 프로그램을 설계할 수 있음.
2. lazy addressing(memory 동적 할당/해제)이 가능함.
3. 여러 프로세스가 동시에 main memory를 사용할 수 있음.

임베디드 등의 매우 간단한 시스템에서는 physical addressing을 사용하기도 하지만, 대부분의 시스템에서는 virtual addressing을 사용함.

각 프로세스들은 main memory를 공유하여 사용하지만, virtual addressing 덕분에 각 데이터가 서로 영향을 미치지 않음. (물론 영향을 미치도록 설계된 프로그램도 존재함.) 만약 physical addressing을 사용한다면 각 프로그램에서 특정 주소의 PA를 사용하도록 코드가 작성이 되어야 하고, 이에 따라 address의 사용이 겹치지 않도록 main memory 사용에 대한 공통적인 규약이 존재해야 함. 하지만 수많은 종류의 프로그램들에 대해 이런 처리를 적용하는 것은 굉장히 비효율적임. 이런 이유로 virtual memory는 computing architecture에서 가장 획기적인 아이디어 중 하나로 여겨짐.



3.1.2. address translation

Definition 114 *주소 번역(address translation)은 virtual addressing에 의해 VA를 PA로 변환하는 것을 말하는데, 이는 MMU가 수행함. MMU(Memory Management Unit)은 VA를 PA로 변환하는 unit임.*

address translation은 memory access 시마다 수행됨.

각 프로세스는 고유한 Memory address space(page table)를 가지고, 이 공간에 현재 사용 중인 PA를 저장함.

하나의 프로세스는 Memory address space(page table)와 Processor state(register, pc 등)로 관리되고, 어떤 프로세스 실행 중에 다른 프로세스를 실행하는 경우 이를 memory에 저장하는 작업을 수행함. 이런 프로세스

사이의 전환을 context switch라고 하는데, memory에 데이터를 저장하고 읽어와야 하므로 빠르게 처리되기 어려움.

address translation은 memory access마다 수행되고, 필요한 경우 main memory에도 접근해야 하기 때문애 성능에 미치는 영향이 크고, critical path에 존재하는 연산임.

3.2. paging

3.2.1. paging

Definition 115 *페이지(page)는 VM에서의 block을 말함. 프로세스의 데이터를 page 단위로 쪼개고, MMU는 각 page별로 PA와 매핑함.*

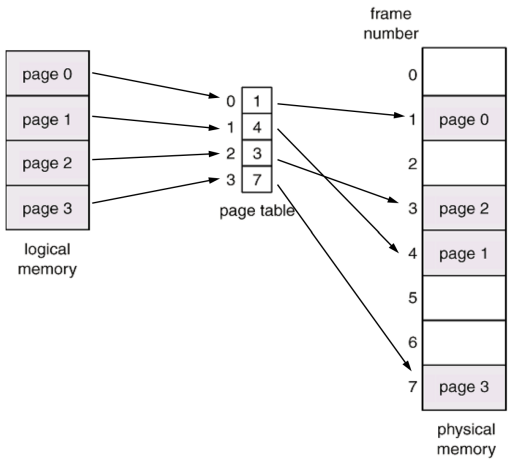
1. page table
page table에 매핑이 정의되어 있음. 각 page는 page number를 가지는데, 이 page number로 page table에서 대응되는 PA를 찾음.

2. page의 크기
page의 크기는 일반적으로 4KB임. VM에서 데이터를 관리하는 단위가 page라면, PM에서 데이터를 관리하는 단위는 frame인데, 대부분의 경우 page는 frame과 크기를 맞춰 정의함. 즉, VM의 page 한 칸과 frame 한 칸이 매핑되어 있는 것임.

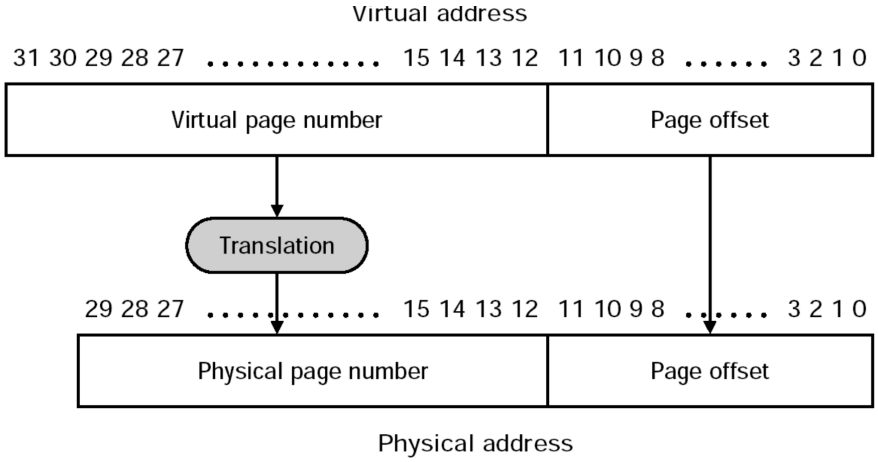
3. address translation
cache에서 mapping 시에 address의 LSB로 block 크기만큼의 block offset을 사용한 것처럼, VA를 PA로 변환할 때 page 크기만큼의 page offset을 사용함. page가 4KB인 경우 page offset의 크기는 12bit임.

VA에서 page offset을 제외한 부분을 tag라고 함. tag를 page table을 통해 PA에 해당되는 부분으로 변환하고, page offset만큼의 부분을 다시 추가하여 PA를 얻는 것. frame과 page의 크기가 같으므로 이런 식으로 변환할 수 있음.

데이터를 작은 고정된 크기로 다루지 않으면 각 프로세스가 사용하는 memory 공간 사이에 비는 공간이 생기기 쉬움. 이를 memory fragmentation이라고 함.



VA는 아래와 같이 page offset을 제외한 부분에 대해 translation되어 PA로 변환됨.



이렇게 얻은 PA를 tag, index, offset으로 쪼개 cache에서 사용함.

이 그림에서는 32비트 VA를 30비트 PA로 매핑하고 있음. 즉, main memory가 1GB 크기인 것. 지금의 변환에서도 알 수 있듯이, VA 개수보다 main memory의 크기가 커지면 매핑이 되지 않아 memory를 전부 사용할 수가 없음. 그래서 64비트 주소 체계가 등장하게 된 것임.

3.2.2. page table

Definition 116 *page table은 각 page과 대응되는 PA의 매핑이 정의되어 있는 table임. page table은 통해 page number에 대응되는 PA를 반환함.*

각 프로세스는 하나의 개별적인 page table을 보유함.

1. valid/invalid
page table은 cache와 같이 valid bit를 사용하고, 추가로 access 권한을 나타내는 bit들도 사용함. 즉, 해당 page에 대한 PA(frame)가 존재하고, 접근 가능하다면 translation을 수행하는 것임.

프로세스는 memory를 동적으로 사용하고, 현재 사용 중인 address는 매핑이 존재하여 page table에 저장되어 있음. 새로 사용하여 frame과 매핑되어야 하는 page에 대해서는 OS가 PM을 할당해 주는 것.

2. PTE
page table에 VA와 PA 사이의 매핑이 존재하는 것을 해당 page에 대해 PTE(page table entry)가 존재한다고 함.

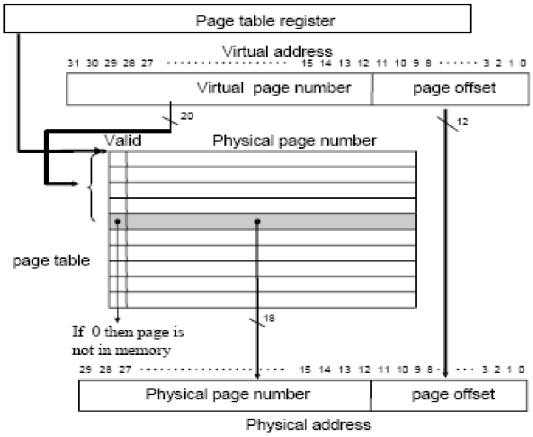
3. page fault
page fault는 address translation에 대한 실패를 말함. 즉, page table에서 page에 대한 access가 실패한 경우인데, page table에서의 miss로 생각할 수 있음.

page fault는 OS에 의해 처리되는데, 새로운 매핑(PA)을 memory에서 가져오고 page table을 수정한 후 명령을 다시 수행하도록 함.

이때 page에 해당되는 부분이 disk에 존재한다면 그 page를 PM에 올리고 매핑을 형성함. 단순히 PM에 데이터를 넣기만 하면 된다면, PM에 메모리를 할당하고 매핑을 형성함.

4. page table의 위치
page table은 main memory에 존재함. page table은 memory access 시마다 사용되어야 하는데 매번 main memory에 접근하는 것은 굉장히 비효율적이므로 TLB라는 cache를 사용함.

page table은 아래와 같은 구조를 가지고 동작함.



3.2.3. TLB

Definition 117 *변환 색인 버퍼(TLB, Translation Lookaside Buffer)는 page table에 대한 cache임.*

memory access 시에는 TLB를 우선 확인하여 hit이면 그대로 address를 사용하면 되고, miss가 나면 MMU는 main memory에 접근하여 page table에서 주소를 가져와 TLB에 반영하고 address를 사용함.

TLB는 대체로 L1 cache보다도 작은 크기를 가지는데, 크기 자체가 작으므로 associativity를 높여도 연산량이 많지 않아 fully associative를 사용하는 경우가 많음. 용량이 작아 빠르고, associativity가 높아 miss rate이 낮음.

TLB도 multi-level로 구성함. L1, L2 TLB는 자주 사용하지만 T3 TLB는 잘 사용하지 않음. 또한 data/instruction에 대해 분리하여 설계하기도 함.

일반적으로 TLB도 replacement policy로 LRU를 사용함.

즉, TLB는 page table에 대해서도 locality를 활용하기 위한 것임.