* 수업자료도 같이 보기 *

[ch1]

Computer revolution

progress in computer technology

- underpinned by Moore's law

Moore's law: 같은 area의 트랜지스터 성능이 2년마다 2배씩 향상된다.

makes novel applications feasible

computers are pervasive

* 딥러닝이나 SNS analysis에는 GPU, NPU 아키텍쳐가 사용된다.
GPU는 그래픽스를 위해 나왔기 때문에 딥러닝에 딱 맞지는 않다.
반면 NPU는 딥러닝에 잘 맞춰져있기 때문에 최근에 많이 사용된다.
GPU: # of small CPUs / NPU: # of ALUs

Classes of computers

Personal computer

- general purpose.
- should be good to any software and applications

Server computer

- high performance
- reliability: 신뢰성이 굉장히 중요하다.
여러 대의 서버 컴퓨터로 큰 규모의 작업을 할 때 하나의 컴퓨터가 crash되면 모든 작업이 잘못되기 때문.

* intel의 i7과 xeon의 차이점
i7: CPU와 GPU
xeon: CPU와 cache
i7과 xeon에 성능 차이는 크게 없지만 reliability 때문에 xeon이 더 비싸다.
=> target application에 따라 i7과 xeon을 각각 썼을 때 속도 차이가 날 수 있다.
computation을 요구하는건 i7, 메모리를 요구하는건 xeon이 빠를 수 있다.

* 우리가 올바르게 만들어야 할 프로그램이 있는데, CPU에서 제대로 작동하고 있는지 확인할 수 없는 경우에 프로그램 동작의 정확성을 높이는 방법
: 여러 번 실행(redundant operation)해서 많이 나온 결과를 도출한다.
-> reliability는 증가하지만 too slow하다는 문제가 생길 수 있음.

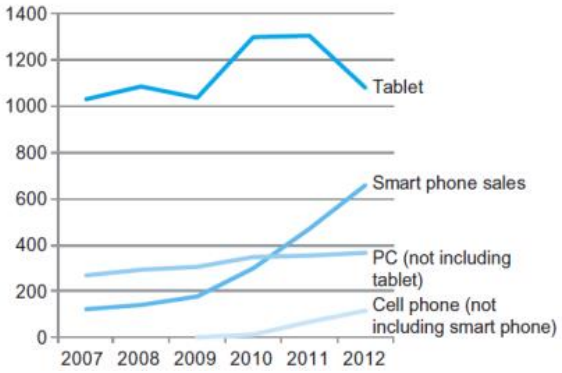
Supercomputer

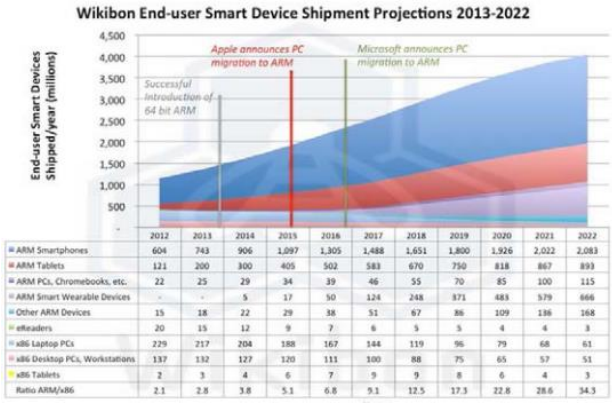
- high performance

Embedded computer

- high energy efficiency
- IoT, Edge device
- Edge computing에서 딥러닝을 해야하는 이유는 security 때문이다.
  딥러닝을 하면 개인정보를 cloud로 보내지 않고 처리할 수 있음.

Processor market





	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022
ARM Smartphones	604	743	906	1,097	1,305	1,488	1,651	1,800	1,926	2,022	2,083
ARM Tablets	121	200	300	405	502	583	670	750	818	867	893
ARM PCs, Chromebooks, etc.	22	25	29	34	39	46	55	70	85	100	115
ARM Smart Wearable Devices	-	-	5	17	50	124	248	371	483	579	666
Other ARM Devices	15	18	22	29	38	51	67	86	109	136	168
eReaders	20	15	12	9	7	6	5	5	4	4	3
x86 Laptop PCs	229	217	204	188	167	144	119	96	79	68	61
x86 Desktop PCs, Workstations	137	132	127	120	111	100	88	75	65	57	51
x86 Tablets	2	3	4	6	7	9	9	8	6	4	3
Ratio ARM/x86	2.1	2.8	3.8	5.1	6.8	9.1	12.5	17.3	22.8	28.6	34.3

- PC와 모바일 중 모바일이 market에 더 많이 팔리고 있다.
- ARM CPU가 x86보다 더 많이 팔리지만 우리가 ARM에서 만든 칩을 볼 수 없는 이유
  : ARM은 라이센싱 CPU 아키텍쳐이다. 애플, 퀄컴, 삼성에서 own chip을 만들 때 ARM CPU를 사용한다. 각 회사의 application processor(chip) 안에 ARM CPU가 들어가는 것.

* 요즘 cloud computing에서 xeon CPU를 ARM CPU로 바꾸려는 시도를 많이한다.
  - GPU가 서버에서 더 중요한 경우가 꽤 있는데, 이런 경우 좋은 CPU가 필요 없기 때문

* cloud에서 하는 작업과 edge에서 하는 작업을 잘 partitioning 하는게 중요하다.

Understanding performance

Algorithm

- 타겟 하드웨어에 잘 맞는 좋은 알고리즘이 중요하다.
  -> fundamental operation을 minimize할 수 있기 때문
- compiler level optimization, better hardware보다 좋은 알고리즘을 사용하는 것이 먼저.

타겟 하드웨어를 제대로 이해하는 것이 중요하다.
: 알고리즘, 최적화 방법은 하드웨어에 따라 달라지기 때문.

ex) CPU는 performing branch에 적합하지 않다. -> branch instruction이 많은 프로그램을 쓰는 경우 CPU 성능이 저하된다. -> compiler level optimization 할 때 branch instruction 수를 줄이는 방향으로 최적화하게 된다. (branch를 줄이는 최적화)

GPU는 bunch of small CPUs -> # of threads가 GPU 성능 향상에 가장 중요하다. -> GPU 프로그래밍의 목적은 to maximize # of threads (thread를 늘리는 최적화)

Programming language, compiler, architecture

- instruction per operation을 minimize해서 performance를 향상한다.

ex) C++은 C에 비해 additional instruction이 있으므로 더 느리다. -> 임베디드 시스템은 효율을 위해 C가 주로 사용된다.

ex) 컴파일러에 따라 최적화 방식이 다르므로 차이가 발생한다. intel 컴파일러는 intel CPU일 때 더 나은 performance를 보여준다.

Processor and memory system

- memory system은 CPU보다 느리기 때문에 많이 느린 memory system을 사용할 경우 빠른 CPU를 사용하더라도 total performance는 느려진다. (CPU는 충분히 빠르기 때문에 CPU별로 큰 차이가 없다.)

I/O system (including OS)

- memory system 설명이랑 비슷하다.

Below your program

application program (C, C++, python 등등)은 hardware에 직접 접속하지 못함

-> system software 필요

system software: compiler + OS

compiler - translate HLL -> machine code

OS - h/w에 접근할 수 있는 interface 제공

Levels of program code

high-level language

- portability (retargetability): 프로그램을 수정하지 않고 target h/w를 쉽게 바꿀 수 있다. 예를 들어 C code를 작성하면 ARM CPU, intel CPU 모두 코드 수정없이 apply 가능하다.
- productivity: 실제 언어와 유사해서 복잡한 프로그램을 작성하기 쉽다
- 컴파일러는 프로그램을 이해하고 CPU, GPU, NPU 각각 최적화를 하므로 좋은 컴파일러 필요

* heterogeneous system

assembly language

- h/w의 specific instruction을 따르기 때문에 target h/w를 바꾸기 어렵다
- polymorphism, OOP 같은 특징이 없기 때문에 HLL 프로그램에 비해 더 효율적이다

* C 프로그램을 작성할 때 어셈블리를 쓰는 이유: 컴파일 된 프로그램이 충분히 빠르지 않거나 새로운 하드웨어가 들어갔을 때 그 하드웨어를 이용하는 어셈블리를 쓸 수 없는 경우, 하드웨어를 잘 안다면 직접 어셈블리를 작성할 수 있다.

* python / C

python: interpreter based (program ability 관점에서 better)

C: compiler based (performance 관점에서 better)

how to improve the python program: C based compiled binary code를 utilize해서 사용할 수 있다

Component of a computer

Touchscreen

resistive: multiple touch 불가능

capacitive: multiple touch 가능 -> more widely used

Opening the box

큰 스마트폰은 큰 배터리를 넣을 수 있어 사용시간이 길어진다.

화면이 커지면 발열이 줄어들 수 있다.

Inside the processor (CPU)

datapath: ADD, MUL, DIV 같은 operation 수행

control: 다양한 instruction을 관리. 더 나은 performance를 위해 control path의 size가 커지고 있지만 선호되지는 않는다

cache memory: 자주 사용하는 data를 저장하는 memory.

* datapath와 control path 중 fundamental하게 더 중요한 것은 datapath이다.
  -> actual operation을 수행하는 part이기 때문

* low end ARM / high end Intel i7
  - datapath의 size는 큰 차이 없음
  - ARM: simple control path -> better energy efficiency
  - Intel: complex control path -> better high abs performance

Inside the processor

apple: multicore

intel 4th: CPU cores, cache memory, GPU

* xeon processor는 GPU 대신 larger cache가 들어가있다.

Abstractions

: low level detail을 없애서 complexity를 줄인다.

instruction set

- abstraction을 통해 CPU를 우리가 원하는 instruction을 수행하는 hardware 정도로 생각할 때 support해야하는 instruction을 모아놓은 set
- hardware developer는 ISA만 잘 지원할 수 있는 h/w를 만들면 해당 ISA를 이용해 만들어지는 s/w를 다 지원할 수 있고, software developer도 ISA만을 이용해 s/w를 만들 수 있다면 모든 해당 ISA를 지원하는 hardware에서 전부 실행할 수 있다.

A safe place for data

volatile memory: DRAM

non-volatile secondary memory: 크고 느림

new memory

- non-volatile / byte-addressible
  -> DRAM(volatile)과 달리 non-volatile / SSD(page단위)와 달리 byte-addressible
- DRAM을 new memory인 RRAM, PRAM으로 대체하려 한다.
  -> DRAM보다는 느리지만 secondary memory보다는 훨씬 빠름
  -> 대체하려면 OS도 바꿔야 한다.
    1) non-volatile characteristic: 현재 OS는 DRAM의 volatile에 맞춰져 있다.
    2) performance: 느린 속도와 높은 용량에 맞는 OS가 필요하다.

Network

요즘은 machine learning이 중요하다. training performance에 중요한 large model에는 multiple distributed PC servers가 필요하다. 여러 개의 서버가 동시에 구동될 때 training performance를 improve하려면 fast network infra structure가 중요하다.

* what is the fast interconnection strategy (data transfer protocol) between GPUs from nvidia?

- nvlink

-> CPU와 GPU 사이에는 PCIE로 연결됨. GPU들 사이에는 nvlink로 연결. nvlink는 PCIE보다 거의 2배 빠르기 때문에 GPU based server를 구현할 때 사용된다. nvlink protocol을 이용해서

multiple GPU를 사용하는 training performance를 향상시킬 수 있다.

Technology trends

한 개의 chip안에 많은 트랜지스터가 들어간다.

DRAM capacity 증가, total performance도 크게 증가했다.

Manufacturing ICs

yield: 결함이 없는 반도체의 비율

architecture 변경으로 yield를 높이는 방법

- CPU에 adder를 추가한다. 실제 product에서는 2개의 adder 중 하나만 쓰지만, 하나가 망가지더라도 다른 adder를 쓸 수 있다면 yield를 높일 수 있다.
- DRAM에 multiple cells이 제대로 동작하지 않는 경우에는 normal cell의 failure cell을 redundant call로 교체한다.

=> technology technique을 바꾸긴 힘드므로 architecture를 바꾼다. (using redundancy)

(단점) CPU의 사이즈가 커진다.

Integrated circuit cost

technology problem이 있을 때 architecture solution이나 compiler level software solution으로 해결할 수 있다.

* Add, Sub, Mul을 수행하는 CPU가 있다. 어떤 CPU가 Mul 연산 결과를 제대로 내지 못할 때 그 CPU로 제대로 결과를 얻으려면 어떻게 해야하는가? (성능은 그다지 중요하지 않은 경우)

-> compiler가 Mul instruction을 없애고 Add instruction을 여러 번 수행할 수 있다.

Defining performance

적절한 performance metric을 고르는 것이 중요하다.

server application - high performance is more preferred metric

embedded systems - energy efficiency can be more preferred in general

Response time and throughput

response time: 하나의 task를 하는데 걸리는 시간 (latency of a single program)

- processor를 바꾸면 response time이 빨라진다
- CPU is good for response time. complex control unit -> single program 수행이 빠름

throughput: average performance

- processor를 추가하면 throughput이 높아진다.
- GPU is good for throughput. multiple low-end cores -> 여러 program을 동시에 수행

Relative performance

두 개의 다른 CPU의 performance를 비교하는 방법

execution time으로 비교할 수 있다.

Measuring execution time

elapsed time: 가장 쉬운 방법이지만 정확하지는 않다.

CPU time: CPU에서 실제로 구동하는 시간만 측정한다.

* CPU time을 정확하게 측정하는 방법

- CPU 내부에 clock information을 check하는 hardware counter가 있어야한다. counter value를 읽으면 CPU time을 정확히 측정할 수 있다. 즉, CPU 내부에 clock information을 저장하는 counter register가 있고, API나 debugging tool을 이용해 counter 값을 읽으면 된다. intel V tune, nvidia visual profiler 같은 runtime profiling tool이 hardware level counter에 접근하는 API를 제공한다.

CPU time

슬라이드 그대로

same CPU with a same frequency를 쓸 때 performance를 향상시키려면 -> reducing number of clock cycles (to execute all the instructions)

CPU time example

슬라이드 그대로

same performance CPU를 만들 때, faster clock을 가지면 (clock frequency가 높아지면) clock cycle이 증가하는 이유?

-> clock frequency를 증가시키면 number of cycle per instruction이 바뀐다. (이해 안되면 강의 영상 1:02:03부터 보기)

Instruction count and CPI

instruction의 cycle이 모두 같다는 가정 하에

CPU time = (Instruction count * CPI) * clock cycle time

= clock cycles * clock cycle time

CPU time을 줄이려면 clock rate는 증가하고, instruction count와 CPI는 줄여야한다.

instruction count를 줄이는 방법

- 효율적인 프로그램 코드

- 효율적인 ISA
- good compiler 사용

average CPI를 계산하는 이유: instruction마다 다른 CPI를 가지기 때문에 CPI가 높은 instruction이 많으면 average가 증가, 적으면 낮아진다. instruction mix에 의해 average CPI가 결정된다.
같은 프로그램이라도 ARM에서 실행할 때와 Intel에서 실행할 때는 서로 다른 CPI를 가진다.

CPI example
동일한 ISA를 가지는 두 컴퓨터의 속도를 비교하려면 CPU time을 계산해서 비교하는 것이 가장 간단하다.
주어진 예시에서 CPU Time(A)가 더 작으므로 Computer A가 더 빠르다.

CPI in more detail
different CPI를 가지는 different instruction class는 같은 CPI를 가지는 instruction끼리 곱한 후 곱한 값을 전부 더하면 total clock cycle을 계산할 수 있다.
예를 들어 ADD와 SUB은 1 CPI, MUL은 2 CPI가 걸린다고 하면 total clock cycle은 1*2 + 2*1이다.
-> 1(cycle) * 2(ADD, SUB 총 2개의 instruction) + 2(cycle) * 1(MUL 총 1개)

CPI example
A가 가장 simple하고, C가 가장 complex하다.
서로 다른 compiler를 사용해 sequence1과 sequence2가 있을 때, sequence1은 total 5 instruction, sequence2는 total 6 instruction을 가진다.
IC만 봤을 때는 sequence1이 더 빨라 보이지만, average CPI를 계산하면 sequence2가 더 빠르다는 것을 알 수 있다. 즉, IC만 보면 안되고 total clock cycle을 계산해야한다.

CPI (MUL: 10 cycle, ADD: 1 cycle, Shift: 1 cycle)일 때 "c = a*3" code의 performance를 향상시키려면 곱셈을 add 또는 shift로 바꿔야하는데 이것이 좋은 compiler가 하는 역할이다.

Performance summary
CPU time을 계산해 performance를 비교할 수 있다.
'performance depends on'은 슬라이드 내용 그대로

Power trends
reducing power가 중요하기 때문에 cloud server system은 cooling expense 때문에 바다나 호수 같은 물 근처에 위치한다.
제시된 그래프는 Intel CPU의 power consumption 그래프이고, 2004년 prescott 이후 core2

architecture를 사용해 power consumption을 줄였다.

power consumption을 줄이려면 capacitive load, voltage, frequency를 줄여야한다. prescott은 high frequency 때문에 power consumption이 증가한 것으로, frequency가 커지면 필요한 register가 많아지고, capacitive load가 커진다.
capacitive load를 줄이려면 chip의 size를 최소화해야한다.

Reducing power, Uniprocessor performance
같은 frequency, area, voltage일 때 performance를 increase하려면 number of core를 increase하면 된다.

Multiprocessors
baseline program은 보통 single thread로 쓰이기 때문에 multicore를 활용할 수 없다.
multi thread를 쓰려면 프로그래머가 multi thread 코드를 작성하거나, 컴파일러가 지원해준다.
parallel programming이 어려운 이유
- load balancing: thread 간의 속도 차이가 있다면 일찍 끝난 thread는 늦게 끝나는 thread가 종료될 때까지 wait하므로 total performance가 fully improve될 수 없다. thread가 동시에 종료되도록 load balance를 맞춰줘야한다.
- synchronization: 두 개의 thread가 같은 data에 접근하려 할 때 correctness를 위해 필요하다.

SPEC CPU benchmark
performance measuring을 할 때 metric을 고르는 것만큼 실행될 target program도 중요하다. 모든 CPU는 각각 다른 structure를 가지고, 어떤 target program에 대해 아주 efficient하거나 그렇지 않을 수 있기 때문에 CPU comparison에서 SPEC benchmark가 많이 사용된다.

Pitfall: Amdahl's law
optimization이 affect된 time (T_affected)는 빨라지지만, 그렇지 않은 T_unaffected는 그대로이다.
-> 빨라지지 않는 부분이 존재하므로 performance를 얼마 이상 빠르게 improve 할 수 없다.
Amdahl's law를 이용해 프로그램을 최적화하려면 hot code region (loop; 여러 번 반복되는 code) 최적화에 집중해야한다. 요즘은 loop이 워낙 빨라졌기 때문에 sequential code 부분을 빠르게 하려고 한다.

Fallacy: low power at idle
A는 긴 코드 한 번, B는 짧은 코드를 1000번 도는 경우
CPU는 B를 최적화하려한다.
반면 GPU는 코어가 여러 개이므로 B를 한 번에 처리할 수 있어 A를 최적화하려 한다.

단, B 코드가 GPU에 하나씩 mapping되려면 loop는 iteration dependency를 가지면 안되고, load balance를 맞춰야한다.

요즘은 dynamic power consumption(power at program execution)이 작아지고 static power consumption (power at idle)이 커져서 static도 최적화가 필요하다 (battery time maximize). 여전히 dynamic이 더 중요하긴 하지만 static도 중요하다는 것을 알아둘 것.

Pifall: MIPS as a performance metric

MIPS는 컴퓨터 사이의 ISA differences, instruction 사이의 complexity differences를 고려하지 않는다.

100 MUL + 100 ADD를 Intel CPU에 넣으면 100개의 instruction(MULADD 100개)이 필요하다.
ARM CPU는 200개의 instruction(MUL 100개, ADD 100개)이 필요하다.

-> instruction이 더 많은 ARM이 더 빨라보이지만 사실 둘은 동일하다. 즉, MIPS는 좋은 metric이라고 할 수 없다.

[ch2]

Instruction set

CPU1과 CPU2가 같은 instruction set을 가지는 경우 동일한 binary code를 두 군데 모두에서 사용할 수 있다.

CPU1(Intel), CPU2(Intel), CPU3(ARM)일 때 CPU1과 CPU2에서는 가능, CPU3에서는 불가능하다. -> CPU3은 다른 instruction set을 지원하기 때문.

같은 instruction set을 지원하는 CPU라도 컴퓨터를 새로 샀을 때 OS를 새로 설치해야하는 이유는 instruction set과 관계없이 컴퓨터의 구성 하드웨어가 다르기 때문이다.

intel i7 CPU 중에서 2015 i7과 2021 i7의 지원하는 instructions의 개수는 다르다. 2021 버전이 더 많은 instruction을 지원한다. 더 많은 instruction을 지원한다는 것은 새로운 하드웨어가 추가됐고, 하드웨어를 이용하기 위한 instruction이 추가된 것이다.

- 어떤 binary code(compiled on 2015 i7)가 있을 때 그 binary는 2021i7에서 execute 가능하다.
- 어떤 binary code(compiled on 2021 i7)가 있을 때 그 binary가 2021에서만 지원하는 instruction을 쓴다면 2015에서는 execute 불가, 아니면 가능하다.
- binary1(compiled on 2015 i7), binary2(compiled on 2021 i7)가 있고, 둘 다 2021에서 execute 가능할 때 일반적으로 binary2가 더 빠르다. binary2는 새로 추가된 instruction을 이용할 확률이 높기 때문.

The MIPS instruction set

ARM instruction과 유사하다.

RISC vs CISC

RISC

- small size instructions
- power efficient (small hardware)
- strong compiler support is essential
- power efficient, but slow

  ; CISC에서 MULLADD로 한 번에 되는 일에 2개의 instruction이 필요하기 때문에 slow

CISC

- complex instructions
- faster
- large hardware

요즘에는 RISC가 more widely하게 사용된다.

최근 intel hardware를 RISC machine으로 볼 수 있는 이유: recent intel CPU 내부에 RISC hardware(simple ALU)가 있고, simple instructions를 execute한다. 하지만 intel은 여전히 x86 instruction (CISC instruction)을 사용하는데, input으로 CISC instruction이 들어가면 instruction buffer에서 RISC instruction(microcode)으로 translate한다.

Arithmetic operations

add a, b, c (sources: b and c, destination: a)

Simplicity favours regularity

makes implementation simpler -> high energy efficiency (simple hardware)

higher performance (highly efficient code) -> strong compiler support is required

Arithmetic example

C code to MIPS assembly code

연산을 위해 총 3개의 instruction이 필요하고, 하나의 instruction은 2개의 operand밖에 가지지 못하므로 temporary register가 필요하다.

// CISC는 ADDADDSUM r1, r2, r3, r4, r5로 연산 가능 -> temporary register 필요x

Register operands

(small, fast) <----------> (large, slow)

CPU (register file) - cache - main memory (DRAM) - storage (SSD)

operand는 location of the data를 의미한다. register file이나 main memory에 있는 data에 접근할 때는 각각 다른 operand를 가진다.
cache는 DRAM의 주소를 사용하기 때문에 cache operand가 없다. 즉, cache의 data는 operand를 지정하지 않는다. storage에 있는 file은 OS의 file system을 이용해서 handle하므로 operand가 없다.
=> we need to care about the data in the register file and DRAM

MIPS hardware에는 temporary value와 saved variable을 저장하는 특별한 assembler name이 있다.

Register operand example

```
function call process
  main( ) {
       add( );
  }
  add( ) {
       ....
  }
```

main, add function 모두 CPU에 있는 register file의 register를 이용한다. add function call을 하고, instruction을 running하려면 그 전에 main function에서 사용하던 register를 어딘가에 저장해두어야한다. 마찬가지로 add function이 끝나면 saved register를 다시 main function으로 restore해야한다. 이 때, 32개의 register를 모두 저장하려면 매우 느리다.

(C language level solution) function inlining이 register save problem을 해결할 수 있다. function call이 생겼을 때 compiler는 function call, return instruction을 remove하고 function을 straight line code로 running해서 total program이 더 빨라질 수 있게 한다. -> function call, return overhead, data transfer overhead (back up the registers to memory)를 remove해서 빨라진다.
* 항상 inlining을 하지는 않는 이유: side effect 때문. hardware에는 적은 register가 있는데 inlining을 하려면 많은 data를 오래 저장하고 있을 register가 많이 필요하고, 그렇게 되면 performance가 감소한다.

(hardware level (ISA-level) solution) function call이 생겼을 때 only saved registers back up to the memory -> back up할 register 개수를 줄일 수 있다. compiler support가 매우 중요하다.

Memory operands
main memory는 composite data를 저장한다.

memory에 있는 data를 이용한 operation을 하려면 arithmetic operation은 register operands만 사용하므로 CPU는 data를 memory에서 register로 보내야한다. 이 때 memory에서 register로 value를 보내는 것을 load, register에서 memory로 result를 저장하는 것을 store라고 한다.
byte addressed: 각 address는 8-bit data를 포함한다. word data를 저장하려면 4addresses가 필요하므로 address는 multiple of 4가 되어야 한다.
MIPS는 big endian, ARM은 little endian을 사용하지만, intel CPU에서 저장한 data를 ARM CPU가 꺼낼 때 endian이 맞지 않으면 문제가 생길 수 있어 최근 ARM은 little, big endian 둘 다 사용가능하다.

Memory operand example 1
main memory에 있는 array data, g, h는 각각 register에 mapping된다. (A는 배열의 시작주소가 맵핑)
&A[8] = &A[0] + 4*(word size)*8(index) => 32($s3)

Memory operand example 2

Registers vs memory
register는 memory에 비해 접근 시간이 빠르기 때문에 compiler는 register를 최대한 많이 사용하려 한다. -> register optimization이 중요하다.
만약 register에 저장될 수 있는 small variable의 수가 register의 수보다 많다면, 자주 사용되는 variable을 우선적으로 register에 저장한다.

Immediate operands
constant data를 저장하는 operands (constant value in(embedded) an instruction)

add (for R-format), addi (for I-format)
addi $s3, $s3, 4 // s3 = s3+4

immediate operand는 maximum value가 2^32 - 1만큼 될 수 없다. constant value는 instruction 안에 포함되고, instruction에는 opcode같은 다른 정보들도 포함되기 때문에 immediate operand가 full 32bit를 사용하지 않는다. 대부분의 경우에 immediate value는 작기 때문에 가능하다.

MIPS에서 register operand를 표현하기 위해서는 5bit가 필요하다. (register0 ~ register31(2^5 -1) 개의 register)

2^16 - 1보다 큰 larger constant를 사용하려면 addi를 여러 번 사용하고, 위의 bit로 옮겨주기 위

한 한 번의 shift가 필요하다.

Unsigned binary integers

2s-complement signed integers

2s complement number를 만드려면 -> invert all the bits and + 1
MSB는 sign bit로 사용된다.

Signed negation

integer value의 ADD만 support하는 CPU0와 ADD, SUB를 support하는 CPU1이 있을 때 두 CPU의 ALU size를 비교하면 얼마나 차이가 날까 -> ALU1이 ALU0보다 약간 크다. SUB를 하려면 2s complement logic만 추가로 필요하기 때문에 둘은 크게 차이나지 않는다.

Sign extension

작은 bit 수로 이루어진 숫자를 더 큰 storage에 저장할 때 sign bit로 남는 공간을 채운다.
immediate operand(16bit)를 register file에 저장하거나 어떤 계산에서 32bit로 써야할 때 32bit data로 변환해야하기 때문에 sign extension이 필요하다.

Representing instructions

instruction encoding: create the 32bit instruction binary from the assembly instruction
register operand는 5bit (because of total 32 number of registers)
ARM CPU는 register가 16개이므로 operand에 4bit가 필요하다.

MIPS R-format instructions

3개의 operand(rs, rt, rd)가 15bit를 차지한다.
operation을 표현하는 op와 funct이 12bit,
shift operation에 사용되는 shamt가 5bit를 가진다.

maximum shift amount가 31(2^5 - 1)이므로 shift amount에 5bit가 할당된다.

R-format example

Hexadecimal

MIPS I-format instructions

immediate arithmetic을 표현할 때 I-format instruction을 사용한다.

거의 대부분의 instruction이 R-format과 I-format으로 matching되도록 해서 format을 최대한 비슷하게 유지해 hardware를 simple하게 만든다.

Stored program counter

instruction과 data 모두 DRAM에 저장된다.

어떤 intel CPU에서 만들어진 binary가 다른 intel CPU에서 동작하려면 두 CPU가 같은 ISA를 가지고 있을 때 가능하다.

(Intel base)x86 system과 ARM based system이 있을 때 cross compiler를 사용하면 ARM binary를 x86에 cross compile 가능하다. cross compile을 하는 이유는 compilation speed를 maximize하기 위해서이다. x86 system은 ARM이나 mobile보다 compile하는 시간이 빠르고, intel이 high memory capacity를 가지고 있어 (속도가 빠르고 memory가 많음) cross compile을 한다.

Logical operations

MIPS instruction 중 shift operation sll과 srl은 shift left(right) 'logical'을 의미한다.

Shift operations

shamt가 5bit인 이유는 max amount가 31(2^5 - 1)이기 때문이다.

logical shift 외에 arithmetic shift는 right shift를 할 때 sign bit를 추가한다.

AND operations

3 operands (R-format)를 사용한다. 어떤 range만 살리고 나머지는 0으로 바꿀 때 mask bits와 AND operation을 한다.

target: $t0 / input: $t1, $t2

OR operation

3 operands (R-format) 사용. 어떤 range는 1로 바꾸고, 나머지는 그대로 유지한다.

NOT operation

1은 0으로, 0은 1으로 compliment한다. MIPS나 ARM CPU hardware는 NOT instruction을 지원하지 않아 NOR 3-operand instruction을 사용해 구현한다. 즉, NOT operation은 pseudo operation이다. regularity를 없애지 않고, 하드웨어의 효율성을 높여 hardware를 간단하게 하기 위해 NOT을 지원하지 않는다.

Conditional operations

원래는 instruction을 sequential하게 수행하다가 loop, if-then-else 같은 instruction을 만났을 때 run time 중에 execution procedure를 바꿔야하는데 이 때 conditional operation을 사용한다.

conditional operand를 사용하면
1 reuse the instruction
2 skip several instructions
3 (otherwise) operand가 NOP으로 change해서 jump를 하지 않고 다음 instruction을 그대로 수행한다.
j: unconditional jump (always jump to the target instruction)

Compiling if statements

if의 condition을 check하고 then으로 갈지, else로 갈지 정할 때 branch instruction이 필요하다.
MIPS code에서 bne $s3, $s4의 comparison result에 따라 Else 또는 next instruction(change bne to NOP)으로 결정된다.
original code는 i와 j가 같은지 비교하는데, MIPS code에서는 bne(not equal)를 사용한다. 그 이유는 better debugging을 위해 original code order(f=g+h 다음 f=g-h)를 유지하기 위해서이다.
'j Exit' code는 Else option을 건너뛰기 위해 존재한다.

Compiling loop statements

&(save[i]) = &(save[0]) + 4*i = $s6 + $s3*4 = $s6 + $s3<<2
multiplication보다 shift가 memory address calculation에 optimize되어있으므로 save[i] 주소 계산에 sll을 사용한다. ($t1에 save[i]를 저장한다.)

C code (save[i]==k)에서는 iteration condition을, MIPS code (bne t0, $s5, Exit)에서는 exit condition을 check한다.

forward branch: for skip several instructions (bne $t0, $s5, Exit ;Loop 종료)
backward branch: for reuse the previously executed instruction (j Loop ;Loop으로 되돌아감)
assembly program을 봤을 때 backward branch가 있다면 원래 프로그램에 loop가 있다는 것을 의미한다.

* forward branch, backward branch의미, 메모리에 있는 데이터를 가져올 때 코드가 만들어지는 과정 알기

Basic blocks

basic block: group of sequential instruction (instructions with a same execution flow), unit of compiler optimization
basic block의 시작 위치: target of branch
basic block의 end point: branch instruction

-> branch target으로 시작하고 branch instruction으로 끝이 난다.

Review: branch

loop는 hot code region(optimize를 해야하는 부분)이므로 backward branch를 보고 loop을 찾는 게 중요하다.

Reviews: procedure call

```
main( ) {
   ...
   add( );
   ...
}
add( ) {
   ...
}
```

main(caller) 중간에 add(callee)를 수행한다.
add execution이 끝나면 main으로 return해서 main function execution을 resume한다.

add를 실행하기 전에 function call (jump to location of first instruction of the add instruction)이 필요하다. function call을 하려면 알아야 할 것
(1) location of the first instruction of add( ) (target function)
(2) return address (location of next instruction of add function call)

function call을 수행하는 instruction에 (공간의 제약 등으로) 하나의 address만 들어가야하는 경우에는 target address의 시작 주소가 embedded된다.
function call을 perform할 때 target address는 instruction에 embedded 되어있으므로 알 수 있고, return address는 function call의 next instruction이므로 automatically하게 알 수 있다. 이 때 function call은 (1) jump to target (2) save the return address to some storage 2가지 동작을 하는데, return address를 stack에 저장해두기 때문에 나중에 end of execution에 stack에서 return address를 꺼내 사용할 수 있다.

Procedure calling (calling convention)
function call이 생겼을 때 program이 해야 할 step
3. acquire storage -> storage는 register와 memory를 둘 다 의미한다. step3는 'to allocate some space to variables of function'을 의미한다.

```
main( ) {
   ...
   d = add(e, f);
   ...
}
int add(int a, int b) {
   int c;
   c = a+b;
   return c;
}
```

1 main 함수의 e, f는 register에 저장된다.
   -> caller execution에서 사용하던 값은 callee function으로 이동하면 그 value를 사용하지 못하므로 저장해둔다.
2 transfer control to procedure (function call)
3 storage는 local variable을 말한다. (add function의 local variable c)
4 perform
5 save result
   -> caller로 돌아가기 전에 result의 값을 register에 저장하는 이유는 add function에서 main으로 return하면 add의 local variable c에 접근할 수 없기 때문이다.
6 return

Register usage

register의 level을 나눠서 사용하고, function call이 생겼을 때 register 전부를 memory에 저장하는 것이 아니라 only subset of registers만 저장한다.
보통 $t0-$t9 register는 memory에 저장하지 않는데, 저장해도 performance 속도 저하 외에 다른 문제는 없으므로 상관없다.
$a0-$a3에는 총 4개의 arguments만 저장할 수 있는데, parameter가 5개인 경우처럼 4개 이상일 때는 register에 저장하지 못한 argument는 memory에 저장된다.

* $s0-$s7을 caller가 아닌 callee가 저장하는 이유: caller가 저장하는 것은 내 data가 중요하니까 저장하는 것이고, callee가 저장하는 것은 내가 사용할 공간을 확보한다는 의미를 가진다.
fundamental하게 다른 점은 없다.

Procedure call instructions

jal instruction: (1) save return address in $ra (2) jump to target address
program counter: register to save location of next instruction to execute

return address가 stack에 있을 때보다 register file에 있을 때 access time이 더 짧기 때문에 return address를 register file에 저장한다. nested code의 경우에 multiple return address with correct order가 필요하면 register file에 저장하는 것이 불가능하며, 이 때는 stack을 사용해야한다.

Leaf procedure example

4개의 parameter(g, h, i, j)와 1개의 local variable (f)

parameter -> register에 저장

local variable을 $s0에 저장하려면 이전에 사용하고 있던 $s0에 들어있던 값을 stack에 저장해두어야한다.

Leaf procedure example

jal이 없는 이유는 jal 이후 코드인 callee function이기 때문이다. (caller에서 jal leaf_example해서 jump해 온 상태)

caller에서 사용하던 $s0를 stack에 저장한다. (DRAM에 있는 decreasing stack -> addi -4해서 공간을 확보하고 저장) -> lw, addi는 stack에서 pop한다.

C는 어떤 function이든 단 하나의 return 값을 가지고, MIPS code에서 $v0가 return 값을 저장하는 레지스터이다.

register utilization을 더 하려면

-> sub $s0, $t0, $t1

add $v0, $s0, $zero 이 두 instruction을

sub $v0, $t0, $t1으로 나타낼 수 있다.

=> $s0를 사용하지 않으므로 push, pop operation이 필요없어진다.

코드 최적화 순서

compiler create assembly code -> compiler level optimization -> compact, efficient code

Non-leaf procedures

다른 procedure를 호출하는 procedure (nested-call)

caller가 stack에 저장하는 것

- return address ($ra)
- arguments ($a0 ~ $a3)
- temporaries ($s(x))

Non-leaf procedure example

argument(a0), return address(ra)는 stack에 저장하고, result(v0)는 저장하지 않는다. (result는 나중

에 필요없는 값이라 저장하지 않음)

MIPS code

* stack operation으로 data를 다루는 방법을 알아야한다.

fact 함수를 호출하면 caller's return address and arguments를 stack에 저장한다.

function call을 수행할 때마다 address는 sp에서 8을 빼고 data를 save한다. (return address와 argument 총 2개 item 저장)

코드를 4부분으로 나눌 수 있다.

(1) addi ~ beq (stack operations - push 포함)

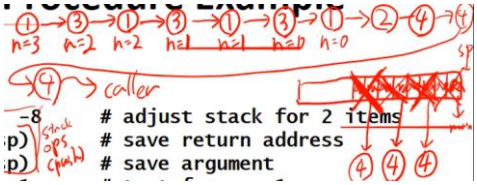
(2) addi ~ jr (stack ops - pop 포함)

(3) addi ~ jal

(4) lw ~ jr (stack ops - pop 포함)

initial n=3일 때 execution (괄호 안은 각 call의 result)

1(n=3) -> 3(n=2) -> 1(n=2) -> 3(n=1) -> 1(n=1) -> 3(n=0) -> 1(n=0) -> 2



Local data on the stack

stack은 function call을 수행하는 동안 update된다.

fp (frame pointer): previous stack pointer before function call (stack의 base address)

fp와 sp 사이 공간을 procedure frame이라고 부른다.

* fp가 필요한 이유: callee function을 수행하는 중에 어떤 dynamic things(register file이 부족해서 stack에 저장하는 등의 경우)에 의해 stack pointer의 값이 바뀔 수 있기 때문에 필요하다.

Memory layout

text: 코드

static data: 전역 변수

dynamic data: heap

stack: automatic storage

stack과 dynamic data는 다른 방향으로 grow한다.

Character data

MIPS architecture에서는 32bit character data가 default로 사용된다.

Byte/Halfword operations

byte(8bit), halfword(16bit)

32bit architecture를 쓰더라도 smaller bit data를 operate하는 경우가 많다.

lb, lbu, sb (load / load ~ unsigned / store byte)

lh, lhu, sh (load / load ~ unsigned / store halfword)

C code에서 target data의 data type이 load, store의 data size를 결정한다.

int a[10]에서는 lw, char b[10]에서는 lb.

word의 address는 0, 4, 8, 12, ... 4씩 증가한다.

character는 0, 1, 2, 3, ... 1씩 증가한다.

short는 2씩 증가

String copy example

예제에서 알아야할 것

1. lb, sb will be created
2. x[i], y[i]의 주소를 계산할 때 index i에 4를 곱하지 않아도 된다.
3. while phrase의 assembly code를 만드는 방법

MIPS code

lbu, sb를 통해 character array data인 x와 y에 접근한다.

character array의 각 data는 1byte이기 때문에 index i의 real address를 계산할 때 4*i(shift by 2operation)를 할 필요없다.

32-bit constants

16bit constant인 61을 32bit로 쓰려면 lui $s0, 61

-> high 16bit에 61을 채움(왼쪽부터). low 16bit는 0으로 채운다.

Branch addressing

32bit instruction 중 target address(32bit)를 저장하기 위한 공간은 16bit밖에 없다.

MIPS에서 PC-relative addressing을 사용해서 target address를 계산

instruction address는 hexadecimal로 0, 4, 8, C, 10, ...이 되는데, 이 값들은 binary에서 0000, 0100, 1000, 1100, 10000, ... 으로, 하위 2bit는 항상 2이다. -> 하위 2bit는 따로 저장하지않고 target address를 계산할 때 shift operation으로 00을 붙여준다.

=> Target address = PC + offset*4에서 (offset*4)의 의미는 'insert last 2bits of 0'

Branching far away

branch target의 range는 current instruction 근처이다. (+2^17 ~ -2^17) (17=16-1+2)

target address가 멀다면 branch instruction으로는 불가능.

-> multiple branch instruction을 사용해서 멀리 있는 branch target으로 갈 수 있다. (execution time은 늘어난다.)

Jump addressing

26bit를 address에 쓸 수 있고, LSB는 00이므로 총 28bit를 표현할 수 있다. -> 4bit만 더 표현하면 된다.

나머지 4bit는 PC의 상위 4비트에서 가져온다. PC 값을 이용한 계산이 아니라 그냥 가져오기만하므로 PC-relative라고 부르지 않는다.

PC(31...28):address*4 = high4bit:address:00

Target addressing example

bne: skip several instructions (forward branch)

j: reuse the previous code again (backward branch)

80012: branch instruction

target address는 80024, current PC는 80016

difference = 80024 - 80016 = 8

offset = 8/4 = 2

-> offset이 positive인 것은 forward branch를 의미한다.

80020: jump instruction. directly pointing the target address (80000)

difference 계산 없이 80000/4 = 20000을 offset에 저장한다.

Addressing mode summary

data가 여러 곳에 있을 수 있기 때문에 다양한 addressing mode를 제공한다.

Immediate addressing

Register addressing

Base addressing

PC-relative addressing (branch)

Pseudodirect addressing (jump)

Synchronization

data race condition을 없애려면 lock process가 필요하다.

lock을 하려면 hardware의 support가 필요하고,

CPU는 atomic read/write memory operation이 제공돼서 lock process를 할 수 있다.

Assembler pseudoinstructions

pseudoinstruction: CPU hardware에 implemented 되어있지 않은 instruction

not은 nor로, move는 add로, blt는 slt와 bne로 구현된다.

pseudoinstruction을 사용하는 이유:

1. (hardware efficiency) hardware에서 너무 많은 instruction을 제공하면 hardware의 size는 커지고, power efficiency가 줄어든다.
2. (instruction format efficiency) 32bit instruction format에서 opcode의 bit 수가 적기 때문에 total instruction의 수를 줄여야한다.

skip한 슬라이드 중에 코드는 잘 보기

Effect of compiler optimization

compile할 때 optimization option을 사용하면 redundant instruction이 줄어들어 performance가 2배 이상 향상된다.

Fallacies

1. powerful instruction을 사용하는 것보다 small instruction과 good compiler를 사용하는 게 performance 향상에 더 좋다.
2. instruction set은 계속 바뀌고, support하는 instruction은 점점 많아지고 있다.

Concluding remarks

[ch4]

Introduction

CPU performance에 영향을 주는 요인

- instruction count: ISA와 compiler에 의해 determine
- CPI and Cycle time: instruction count가 같더라도 CPU에 따라 달라진다.
  better CPU -> better CPI and cycle time

Instruction execution

CPU가 instruction을 execute하기 위해 수행하는 5가지 job

1. Fetch: PC가 가리키는 instruction memory의 위치에서 그 값을 가져오는 과정 (IR(instruction register)에 저장)

2. Decode: CPU try to understand every meaning inside the instruction

-> CPUs create the control signals depending under current instruction

-> if the instruction need to read register file data, CPUs will perform register read.

(decode + register read)

3. Execute: create some data using ALU

- create arithmetic result for normal instruction
- calculate memory address for load/store instruction
- branch instruction -> 보통 branch instruction에서 main ALU는 comparison에 사용
  target address를 계산하는 ALU는 따로 있음
  => 2 ALUs are required for branch

4. Memory (if required): process to access memory for getting data for load or saving data for store

5. Write-back: to save resulting data to the register file and updating the PC(target address or PC+4(next instruction in code)) (register write + PC update)

CPU overview

어떻게 구성돼있는지 확인

Multiplexers

표시된 부분에서 두 data가 합쳐질 때 다른 값이면 data conflict 발생

-> multiplexer with selection bit로 결정 (selection bit은 control path에서 instruction에 따라 결정)

Control

mux가 추가된 회로.

arithmetic instruction에는 data memory가 쓰이지 않고, non-branch instruction에는 두번째 adder가 사용되지 않듯이 instruction에 따라 필요한 resource가 달라진다.

필요없는 resource는 끌 수 있도록 control signal을 사용하는데, control signal은 decode stage에서 만들어진다.

Logic design basics

combinational element: input에 따라 data가 바로바로 바뀌는 element

combinational logic은 logic gate

sequential (state) element: input에 따라 output이 항상 바뀌지는 않는다. some special condition일 때 update된다.

sequential logic은 FF이나 register

Building a datapath

datapath: elements that process data and addresses in the CPU

refining the overview design by adding some more details

Instruction fetch

1. read instruction
   - using PC value. PC값을 reading address로 넣으면 instruction을 read할 수 있다.
2. update PC value to the next instruction
   - adder를 이용해 4만큼 증가

R-format instructions

decode stage

register file에서 2개의 register operand를 읽어 arithmetic/logical operation 수행 (memory operation은 x)

3개의 register port 필요: read1, read2, write (각각 5bit)

3개의 data port 필요: 1 input data, 2 output data (각각 32bit)

execute stage

ALU가 여러 operation을 support하기 때문에 current instruction에 맞는 적절한 operation을 수행하기 위해 ALU operation control signal 필요하다.

Load/Store instructions

memory address, write data bus, read data bus가 필요하다.

load/store instruction의 offset은 16bit signed data -> 32bit data로 바꾸기 위해 sign-extension을 해야한다. -> additional sign-extend logic이 필요하다.

Branch instructions

register operands를 읽고 비교 (main ALU)

   -> compare operation: subtraction instruction without any output data

calculate target address (small adder)

(다음장)

- branch target calculation을 위한 one additional adder
- sign-extension logic
- performing branch instruction을 위한 shift logic

adder의 결과로 나온 branch target은 PC에 저장된다.

Composing elements

By having all required hardware logics using multiplexers, we can have the resulting CPU.
multiplexers are required for avoiding the data conflict.

=> **R-Type/Load/Store datapath**

basic thing of CPU
R-type instruction - registers, ALU
load/store - additional data memory, sign-extension logic

Full datapath

hardware for branch instruction을 추가해서 final resulting CPU 구성
branch instruction - shift logic, adder 추가
PC가 정해지는 2가지 방법
- simple adder by adding 4
- branch instruction

ALU control

add, sub, mul, div 등의 다양한 function(general for R type instructions)을 perform할 때 ALU를 사용해야한다.
memory operation에서는 address calculation을, branch instruction에서는 comparison operation을 ALU를 사용해서 수행한다.
이런 function을 사용할 때는 control signal (ALU control bit)을 사용.

(다음 장)

Load/Store: add function (to generate target address)
Branch: subtract operation (for comparison)
R-type: add, sub, logical operations, set-on-less-than
각 instruction의 opcode를 통해 ALU control bit를 정하는 ALU function code가 정해진다.

The main control unit

opcode: target operation을 나타냄
register operands (rs, rt, rd)
- R-type: two read, one write
- load/store, branch: one read, one write (one remaining operand는 immediate value)

Datapath with control

4 CPU structure for both datapath and control path

control path - regdst, branch, memread, ..., regwrite

control signal은 instruction에 의해 만들어진다.

R-type instruction

register file과 ALU가 mainly used, data memory는 disabled

instruction을 보고 ALUOp이 결정되고 ALU control bit에 들어가고 ALU와 연결된다.

regDst control bit가 mux의 selection bit로 사용된다.

ALU의 result는 register file에 다시 연결된다.

Load instruction

data memory is enabled for read operation (write data port는 disable)

ALU is enabled to generate target memory address

only two register is enabled -> 1 read, 1 write => to generate the target address.

2 data port is enabled -> 1 write, 1 read => read port will be transferred to ALU

instruction의 하위 16bit -> sign extension -> 32 bit signed bit => connect to ALU

- ALUSrc가 mux의 selection bit로 들어간다.

register file에서 나온 data, instruction에서 나온 immediate value를 add해서 memory target address(ALU result)를 생성한다. memory data는 data memory data에서 out으로 나가서 다시 register file (rt)에 저장된다.

* R-type instruction과 달리 write regitser에 instruction [20-16]이 들어가는 이유: load/store instruction에서 target address는 rt에 맵핑된다. R-type에서는 (rs, rt, rd), Load/store에서는 (rs, rt) 이므로 write register에 들어갈 것을 적절히 고른다.

Branch-on-Equal instruction

ALU result value is disabled: ALU는 comparison operation으로 여전히 사용되지만 subtraction의 결과를 저장하지 않기 때문에 ALU의 result는 필요 없다. zero bit만 comparison에 사용됨

data memory is disabled: branch instruction에서 memory structure는 사용되지 않는다.

two ALUs are enabled: branch target address를 계산하는 small adder가 추가된다.

=> two addition (one for comparison, one for create target address)

Implementing jumps

unconditional jump instruction

4 + 26 + 2(LSB 00) => 32bit target address 생성 가능

이 때 계산 없이 old PC에서 상위 4bit를 가져오기만 하므로 addition은 사용하지 않음

Datapath with jumps added

additional adder is disabled: addition하지 않음

additional control signals and additional logic is required

: shift left 2 - LSB에 00 추가

shift left 2를 하고 나갈 때 PC+4가 concat되어서 jump address가 만들어진다.

Performance issues

각각의 instruction은 각자 다른 delay를 가진다.

critical path: longest delay를 가진 instruction (load instruction)

- instruction memory -> register file -> ALU -> data memory -> register file
- 모든 component에 접근하므로 delay가 많아져서 느리다.

다른 instruction은 모든 component가 아닌 일부만 사용하므로 load보다 짧다.

synchronous machine에서 clock period는 longest delay에 의해 결정되고, 모든 instruction은 동일한 clock period를 가진다. 이 때, 다른 instruction execution에 processor가 idle할 수 있으므로 문제가 된다.

* store는 writeback이 없이 (fetch -> decode -> execute -> memory)까지만 하므로 longest가 아니다.