System Programming

Computer vs 8-bit adder

옛날에는 계산하는 장치를 컴퓨터라 정의했으므로 Adder도 컴퓨터였다.
그러나 현재 컴퓨터는 데이터를 가공/연산하여 저장하는 장치라 할 수 있고,
Adder은 메모리가 없어서 저장이 안되므로 컴퓨터가 아니다. 지금의 Adder은 컴퓨터의 일부이다.

Turing Machine

- Alan Turing이 발견한, 모든 연산(계산)이 가능한 가상의 기계
- Turing Machine으로 사람이 풀 수 있는 대부분의 계산들이 가능하다.

Von Neumann

- Turing Machine을 폰 노이만이 확장시킨 구조로, 오늘날 컴퓨터들의 프로토타입이다.
- 바꿔 말하면 모든 컴퓨터는 Turing Machine을 기반으로 하고 있다.
- 프로그램을 메인 메모리에 내장하는 방식이다. (Stored Program의 개념 도입)

1. 프로그램을 구성하는 명령어들을 내장 메모리에 순차적으로 배열한다.
2. 조건에 따라 메모리의 특정 위치들에 있는 명령어들을 불러와 실행한다.
3. 데이터 메모리와 프로그램 메모리가 구분되어있지 않고 함께 섞여있다.

CPU, Register, ALU

- CPU(중앙처리장치)는 명령어의 해석과 자료 연산, 비교 등의 처리를 제어하는 장치이다.
- 레지스터는 CPU 내부에서 데이터를 임시로 저장하는 장치로, 속도가 가장 빠른 메모리이다.
- 수많은 레지스터가 CPU 내에 들어있다.
- ALU은 산술 논리 연산 장치이다.

Executed Program in Computer

1. HDD에 프로그램이 Code/Data/Heap/Stack으로 저장되어있다.
2. Memory에 이 프로그램이 Loaded되고, RAM과 CPU사이에서 상호작용한다.
3. 프로그램 시작 전, CPU의 initial state에는 프로그램이 시작하는 메모리 주소를 담고 있다.
4. 코드를 컴파일 하면, 코드 내용대로 명령어 코드가 메모리에 저장이 된다.
   (이 명령어 코드를 정리한 표가 MIPS 코드표)
5. 메모리에서 명령어를 찾아 CPU로 가져온다. (Fetches)
   - Program Counter라는 레지스터에서 다음 명령을 수행할 메모리의 주소를 가리킨다.
   - 메모리 주소 레지스터에서 PC의 주소를 넘겨받고, 그 주소를 찾아가 명령을 가져온다.
6. CPU가 명령어를 해석하고(Decode) 이에 대응하는 하드웨어 작업을 수행한다.
   - ALU에 의한 산술/논리연산, 데이터 이동 Read/Write, 흐름제어 등
7. 명령 실행을 완료하고(Execute), 다음 명령 처리를 위해 Instruction Pointer(PC)를 업데이트한다.

Memory

1. 각 Data Cell은 Single Bit (0 or 1)로 저장된다.
2. 대부분의 컴퓨터에서, 데이터를 저장하는 최소 단위는 1바이트(8비트)이다.
3. 메모리의 주소값(위치)는 고유한 숫자로 지정되어 있다.
4. 모든 Byte에는 고유의 주소를 가지고 있다.

Memory Operations

1. Read (Load)
   - 메모리로부터 데이터를 불러오려면, 메모리의 주소를 알아야 한다. Read(Address)
2. Write (Store)
   - 메모리에 데이터를 작성하려면, 작성할 주소와 작성할 값을 알아야 한다. Write(Address, value)

Memory Type

1. 휘발성 메모리
   - 컴퓨터의 전원이 종료되면, 메모리에 있던 내용도 사라진다. (= 지속적인 전력 공급이 필요)
   - RAM (SRAM, DRAM)
2. 비휘발성 메모리
   - 컴퓨터의 전원이 종료되도, 메모리에 저장된 정보가 유지된다. (= 전력 공급이 필요X)
   - 테이프, ROM, HDD, SSD

   → 요즘에는 RAM의 자유로운 정보입출력과 ROM의 영구저장 장점을 합친 NVM도 있다.

Finding Memory Patterns

From 0x1000:0000 to 0x2000:0000, memory is filled with the pattern
0x0000 0x1111 0x2222 0x3333 0x4444 0x5555 0x6666 0x7777 0x8888 0x9999 0xaaaa
0xbbbb 0xcccc 0xdddd 0xeeee 0xffff 0x0000 0x1111 0x2222 0x3333 0x4444 0x5555
0x6666 0x7777 ... (i.e. repeat from 0x0000 to 0xffff).

From 0x3000:0000 to 0x4000:0000, memory is filled with the pattern
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f
0x00 0x01 0x02 0x03...
(i.e. repeat from 0x00 to 0x0f)

Rest of the memory is filled with 0.

4. Write the value in the address 0x2468ace (single byte value). You can disregard the endianess. (10pt)

   0x02468ace → Rest of the memory is filled with 0 → 0x00

5. Write the value in the address 0x13579bdf (single byte value). You can disregard the endianess. (10pt)

   맨 앞자리 1 (0x1000:0000 to 0x2000:0000) → d는 홀수 (0x88 to 0xff) → f자리 확인 → 0xff

6. Write the value in the address 0x12345678 (single byte value). You can disregard the endianess. (10pt)

   맨 앞자리 1 (0x1000:0000 to 0x2000:0000) → 7은 홀수 (0x88 to 0xff) → 8자리 확인 → 0xcc

RAM

- Random Access Memory로, 사용자가 자유롭게 내용을 읽고 쓸 수 있는 기억장치이다.
- Can access random address → 임의의 위치에서든 똑같은 속도로 접근하여 읽고 쓸 수 있다

DRAM

- 0과 1로 디지털 정보가 표현되고 저장한다.
- SRAM보다 속도가 느리지만 집적이 용이하고 저렴하여 컴퓨터에서 많이 사용한다.
- 축전기로 구성되어 있으며 충전 상태로, 즉 지속적인 재충전을 하며 정보를 기록한다.
(축전기가 시간이 지나면 저절로 방전되기 때문이다.)

SRAM

- DRAM과 다르게 축전기로 이루어져 있지 않고, NAND flip-flop gate를 사용한다.
- 속도가 빠르지만 구조가 복잡하여 집적에 불리하다.
- 지속적인 전력 공급이 필요 없으므로, DRAM과 달리 Refresh를 위한 추가 회로가 필요 없다.
- 캐시메모리에 주로 사용된다.

ROM

- Read Only Memory로, 한번 기록하면 삭제나 수정이 불가능한 기억장치이다.
- 한번 기록한 정보는 전원 유지와 관련 없이 지워지지 않는다.
- 첫 내용 작성(초기화)할 때만 작성이 가능하고, Run-time일 동안은 작성이 불가능하다.

GPU

- CPU가 복잡한 연산을 빠르게 한다면, GPU는 SIMD로 단순한 연산을 다량으로 처리하는 역할이다.
- SIMD : 하나의 명령으로 여러개의 값을 동시에 계산하여 데이터를 얻어내는 방식
- 집약적인 작업에서 유용 : 그래픽 작업, 과학적인 작업(AI 등)

Accessing Memory

- CPU와 메모리 사이에서 Address, Data, Control이 서로 상호작용 한다.
- Control(Read, Write)를 하기 위해 Address와 Data가 필요하다.
- Read(Address), Write(Address, Value)

Cache Memory

- CPU 내부에 있는 메모리로 중요한 데이터를 임시로 저장하여 사용하는 기억장치이다.

※ 대부분 프로그램에서는 Temporal locality와 Spatial locality를 갖고 있다. 이러한 데이터 지역성들을 활용하여 메인 메모리에 있는 데이터를 캐시 메모리에 임시로 불러와 두고, 프로세서에 필요한 데이터를 찾을 때 캐시 메모리에 우선 접근함으로써 더 빠른 속도로 데이터를 찾을 수 있다.

The Principle of Locality

Look at the following two C program routines:

routine A

```
...
for (long long i = 0 ; i < 0x10000 ; i++) {
  for (long long j = 0 ; j < 0x10000 ; j++) {
    array[i] [j] = array[i] [j+1] + 10;
  }
}
...
```

i=0, j=0
array[i][j] arrary[i][j+1]
[0][0] , [0][1]
i=0, j=1
[0][1] , [0][2]
[0][2] , [0][3]

[0,0][0,1][0,2][0,3]...[][0,0x1000][1,0][1,1][1,2][1,3]...[][]
...
□□□□□□□□□□□□□

---

routine B

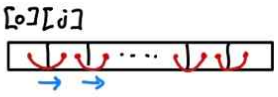
```
...
for (long long j = 0 ; j < 0x10000 ; j++) {
  for (long long i = 0 ; i < 0x10000 ; i++) {
    array[i] [j] = array[i] [j+1] + 10;
  }
}
...
```

j=0, i=0
array[i] [j]
[0][0], [0][1]
j=0, i=1
array[i] [j]
[1][0], [1][1]
[2][0], [2][1]

최근에 접근했던 주소값을 다시 접근하려는 경향

9. Explain the temporal locality presented in routine A. (10pt)

[0][0] : 0x1000 [0][1] : 0x1004 [0][2] : 0x1008
한번 사용한 메모리 주소의 영역을 계속 접근한다.



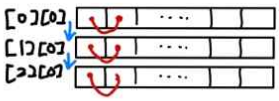
10. Routine A and routine B do the similar job, but has a lot difference in terms of execution time. Which do you think is faster? (5pt)

Routine A

최근에 접근했던 주소값의 근처 주소에 접근하려는 경향

11. Explain the spatial locality presented in routine B. (10pt)

[0][0] : 0x1000 [0][1] : 0x1004
[1][0] : 0x1000 + 0x10000 x 4 [1][1] : 0x1004 + 0x10000 x 4
변수/index 크기
한번 사용한 메모리주소의 근처는 공계가 없는데 접근한 확률보다 더 높다



Temporal locality (시간적 지역성)

- 한 데이터가 엑세스되면 이 데이터가 빠른 시간 내에 다시 엑세스 될 확률이 높다.

예시 for문에서, i = 0, j = 0일 시, arr[0][0]에 arr[0][1] + 10을 할당해주고,
j = 1이 되면 arr[0][1]에 arr[0][2] + 10을 할당해주듯이, 엑세스한 arr값을 반복해서 읽고 사용한다.

Spatial locality (공간적 지역성)

- 한 데이터가 엑세스되면 인접해있는 데이터들도 곧 엑세스 될 확률이 높다.

예시 for문에서, i = 0, j = 0일 시, arr[0][0]에 arr[0][1] + 10을 할당해주고,
i = 2가 되면 arr[1][0]에 arr[1][1] + 10을 할당해주듯이, 엑세스한 arr값의 인접한 arr값을 읽고 사용한다.

Disk Read & Write

1. Disk Arm이 움직이며 목표 Track으로 Arm Head가 이동한다.
2. Spindle Motor가 디스크를 회전시키며 Arm Head 밑에 목표 Sector를 위치시킨다.
3. Arm Head가 현재 위치의 각 Sector들로부터 데이터를 읽기 시작한다.

Direct Memory Access

- CPU의 개입 없이 I/O장치와 기억장치 사이의 데이터를 전송하는 접근 방식

<DMA를 이용한 디스크 데이터를 메인 메모리로 전송하는 방법>

1. DMA가 디스크 제어장치로 데이터 전송을 요청한다.
2. 데이터가 디스크 버퍼에서 메인 메모리로 전송한다.
3. DMA controller가 데이터 전송을 완료하면 디스크 제어장치에서 이를 인식한다.
4. DMA controller가 인터럽트(전기적인 신호)를 만든다.
5. CPU가 인터럽트를 인식한 후 메인 메모리에서 데이터를 읽기 시작한다.

Compiler & Linker & Loader

- 컴파일러 : 소스 코드를 컴퓨터가 인식할 수 있도록 Binary Code로 변환하여 Object file을 생성
- 링커 : 여러개의 Object file을 엮어서 Linking해주며 최종 실행 파일이나 라이브러리 파일을 생성

※ 요즘 프로그램은 컴파일러와 링킹을 한 번에 끝내게 설계되어 있으며, 컴파일러 안에 링커가 있다.

- 로더 : 링커로부터 생성된 파일을 RAM에 로딩하여 CPU가 파일을 실행하도록 도와준다.

Numeric Data

- Signed integer는 음수/양수 값 모두, Unsigned integer은 양수 값만 가진다.
- Signed integer의 범위는 -2^31 ~ 0, 1 ~ 2^31-1 이다.

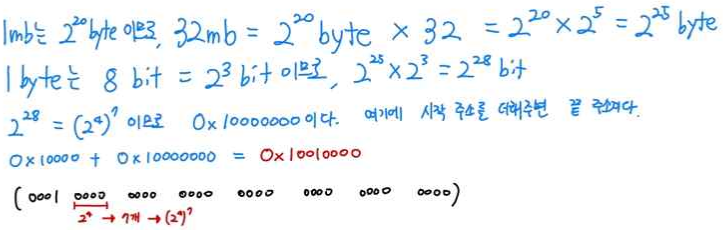
(양수 : 0x0000:0001 ~ 0x7FFF:FFFF, 음수 : 0x8000:0000 ~ 0xFFFF:FFFF)

- 정수 N의 2의 보수는 비트를 모두 flip하고 +1를 해주면 되고, 이 값은 -N이다.

Data Size

- 1 byte : 8 bit
- 1 kb : 2^10 byte
- 1 mb : 2^10 kb = 2^20 byte

8. Think about a storage that the address begins from 0x10000. If the computer uses an address for every single bit, what could be the memory address range for 32MB main memory? Write the start address, and the end address. (10pt)
(e.g. 0x1234, 0x5678)



Bitwise AND/OR, Bitshift

1. AND : 0b1111:1111 & 0b1111:0000 → 0b1111:0000
2. OR : 0b1111:0101 | 0b0000:1111 → 0b1111:1111
3. NOT : ~0x1111 → 0x0000
4. XOR (같으면 0, 다르면 1)
   - 0b1111:1111 ^ 0b1010:1010 → 0b0101:0101
   - 0xaaaa:aaaa ^ 0x1357 → 0xaaaa:aaaa ^ 0x0000:1357 → 0xaaaa:b9fd
5. Shift
   - 0b1111 >> 2 → 0b0011
   - 0x2468 >> 1 → 0x1234
   - 55 >> 2 → 13
   - 55 << 2 → 220

Pointed Numbers in Binary

- 소수점 첫째 자리부터 차례로 0.5(1/2), 0.25(1/4), 0.125(1/8), 0.0625(1/16)로 가중치가 정해져 있다.
- 이 가중치들로 더하며 사용된 자릿수에는 1, 사용되지 않은 자릿수에는 0을 넣어준다.

① 13.75 → 0.75 = 0.5 + 0.25 = 0.11 → 1101.11
② -0.5625 → 0.5625 = 0.5 + 0.0625 = 0.1001 → -0.1001

Floating Point

- 크게 Sign, Exponent, Fraction으로 32bit로 이루어져 있다.
- Sign : 양수 0, 음수 1 / Exponent(지수부, 8비트) / Fraction(가수부, 23비트)

1. 10진수를 2진수로 표현해준다.
2. 2진수에서 맨 앞의 1만 남기도록 소수점을 당겨주거나(^n) 밀어준다(^-n).
3. 당겨준 수에서 원본으로 가기 위한 2^n을 곱해주는 꼴로 나타내준다.
4. 여기서의 n값 + bias가 지수부이다. bias : 2^(지수부의 비트 - 1) - 1 → 8비트는 127
5. 당겨준 수에서 소숫점 이하 모든 숫자들이 가수부이다.

① 1101.11 → 1.10111 x 2^3 → 127 + 3 = **130 : 지수부, 1011100~ : 가수부**
0/10000010/10111000000000000000000
② -0.1001 → -1.001 x 2^-1 → 127 + (-1) = **126 : 지수부, 0010000~ : 가수부**
1/01111110/00100000000000000000000

MIPS Instructions

- MIPS : 명령어 집합 체계
- 컴파일러가 C언어를 MIPS 어셈블리 언어로 바꿔주고, 어셈블러가 Binary language로 바꿔준다.
- 베릴로그 (1) : overflow 예외처리를 해준다.
- 베릴로그 (2) : 상수 부호비트 16bit + 기존 상수 16bit = 32bit로 상수를 확장
- 베릴로그 (3) : 0으로 16bit + 기존 상수 16bit = 32bit로 상수를 확장
- Load Word : 메모리의 주소 값을 레지스터에 저장한다. rt = rs + 상수
- Store Word : 레지스터의 주소 값을 메모리에 저장한다. rs + 상수 = rt
- Add : 레지스터 + 레지스터를 rd에 저장한다. rd = rs + rt
- Addi : 레지스터 + 상수를 rt에 저장한다. 오버플로 예외처리O. (1, 2) rt = rs + 상수
- Addiu : 레지스터 + 상수를 rt에 저장한다. 오버플로 예외처리X. (2) rt = rs + 상수

MIPS Format

- R : 명령어의 길이를 같게 하되, 명령어 종류에 따라 형식을 다르게 한다. (rd - rs - rt)

op : 연산자 / rs : 산술 피연산자1 / rt : 산술 피연산자2 / rd : 결과 피연산자 / shamt : shift 이동량 / funct : 기능

- I : 수치연산과 데이터 전송명령어에 사용된다. (rt - rs)

op : 연산자 / rs : 산술 피연산자1 / rt : 산술 피연산자2 / immediate : 상수

- J : opcode, address 2가지로만 구성되있다.

MIPS Code

1. 레지스터는 r0(00000) ~ r31(11111)로 32개만 있다고 여기서는 간략히 가정.
2. 명령어를 확인하고 이에 맞는 Format을 찾는다.
3. op/func와 피연산자, 상수 등을 참고하여 Format에 맞게 작성해준다.

① add r1, r2, r3

add → r1 = r2 + r3 → Format R → op : 0, func : 20 ⇒ $32_{(16)}$

000000/00010/00011/00001/00000/100000 → 0x00430820

② addiu r29, r29, -8

addiu → r29 = r29 + (-8) → Format I → op : 9

001001/11101/11101/1111111111111000 → 0x27bdfff8

③ sw r29, r30, 0

sw → M[r30 + 0] = r29 → Format I → op : 2b

101011/11110/11101/0000000000000000 → 0xafdd0000

<MIPS 참고자료>

MIPS - 구현

- MIPS에서의 명령어 실행
  - 클럭이 뛴후 레지스터를 읽음, ALU는 바로 출력
  - **PC**를 메모리로 보내고, **Memory**로부터 명령어 가져온 후, 명령어 필드를 보고, 두개의 레지스터를 읽음
  - 레지스터 읽은후, 명령어가 ALU를 사용하는 이유
    - 주소계산 : 참조명령어는 주소계산위해 ALU 사용
    - 연산 : 산술/논리 명령어는 연산을 위해 ALU 사용
    - 비교 : 분기명령어는 비교하기 위해 ALU 사용
  - ALU 사용후 명령어 실행을 끝내기 위한 행동
    - 참조명령어 : 메모리에 접근한후
    - 저장명령어 : 메모리에 접근한후 (데이터저장)
    - 적재명령어 : 메모리에 접근한후 (데이터읽기)
    - 산술논리 명령어 / 적재명령어 : ALU나 메모리에 온 데이터를 레지스터에 쓰고난후
    - 분기명령어 : 다음 명령어의 주소를 갖게한후

MIPS - R형식 명령의 데이터패스 설명

- 명령어 실행은 한클럭 사이클에 일어나며, 4단계로 나눌 수 있음
- 예 : add $t1, $t2, $t3의 R형식 명령어의 데이터패스 동작
  - 1. Instruction을 Instruction Memory에서 가져오고 PC값을 증가시킨다.
  - 2. 두 레지스터 $t2, $t3를, Register file(접근할 레지스터 번호를 지정함으로서 읽고 쓸 수 있는 레지스터들의 집합)로 부터 읽는다.
  - 3. ALU가 연산을 하게 되는데, 연산된 결과를 저장하기 위한 $t1 선택을 위한 과정으로, 명령어의 funct 필드(비트 5:0)의 기능코드를 참조해 ALU의 제어신호를 만든다.
  - 4. Control의 ALUOp 제어유닛의 입력 유닛인 OPCODE(31:26)를 보고 RegDst라고 하는, 멀티플렉스 제어 신호를 출력으로 보낸다. RegDst를 보고, 명령어 비트 15:11인 (ALU의 결과 값이 레지스터 파일에 기록되는데) 목적지 레지스터에 해당되는 $t1를 선택하고 $t2, $t3의 결과값을 $ t1에 저장함

https://blog.naver.com/lws6665/222831809755
https://inyongs.tistory.com/121