

파트1. 하드웨어

이준호

Blog: <https://tot0rokr.github.io>

E-Mail: tot0roprog@gmail.com



개요

01

하드웨어 개론

PC에서 사용되는 부품들이 어떤 역할을 하고 기능을 하는지 간략하게 살펴본다.

02

프로그램 동작

컴퓨터에서 프로그램이 실행되면 어떤 과정을 통해서 cpu가 실행될 수 있는지 알아본다.

03

하드웨어 개념

하드웨어가 동작하는 데 필요한 컴포넌트와 각 컴포넌트의 기능, 각 부품이 사용되는 이유와 역할, 부품 간의 의존관계와 구조 등을 알아본다.

04

가상 시나리오

앞에서 알아본 내용들을 통합하고, 데이터들을 이동하는 모습을 가상으로 시뮬레이션 해본다.

하드웨어 개론

하드웨어의 역할과 기능





하드웨어 핵심편

CPU, RAM, BUS

CPU (Central Processing Unit)

하드웨어 1



- CPU 중앙처리장치
- 연산, 제어 등 모든 “처리”를 담당

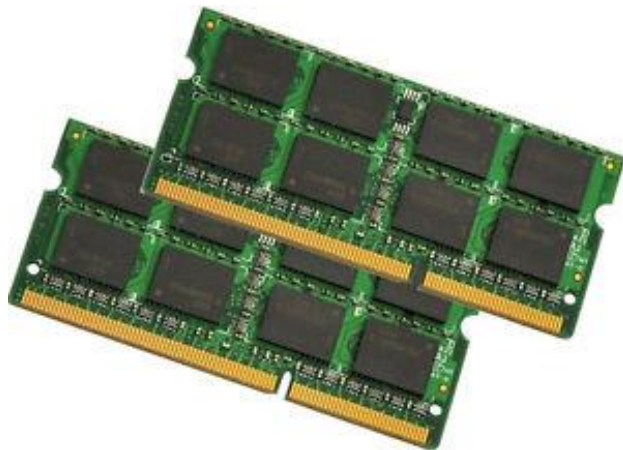
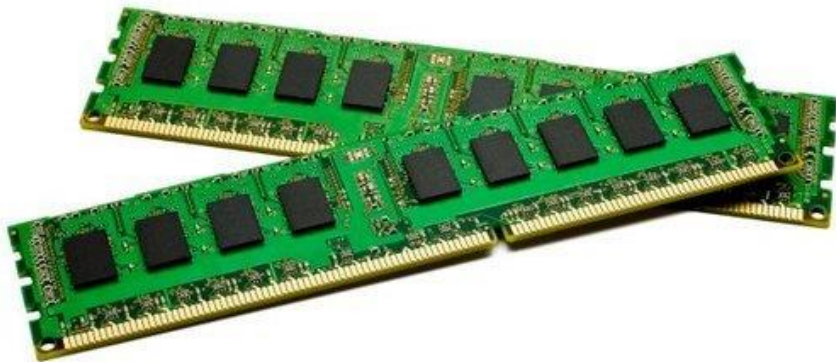


RAM (Random Access Memory)

하드웨어 1



- RAM 임의 접근 기억 장치
- CPU 혹은 기타 장치들이 사용할 데이터를 기억하는 “주 메모리”
- “메모리” 라고 하는 것은 99% RAM을 지칭
- 임의 위치(무작위 위치)에 접근 하는 속도가 모두 동일
- Dynamic Ram, Static Ram

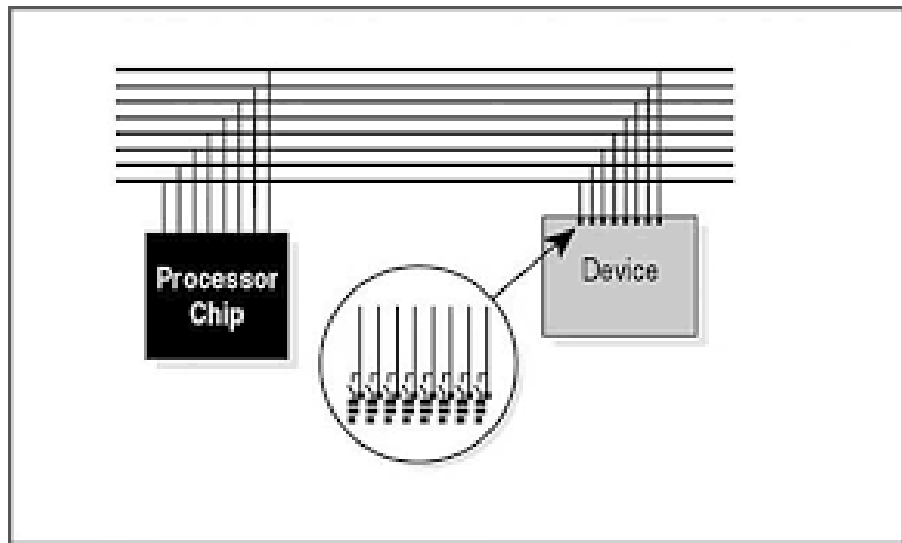
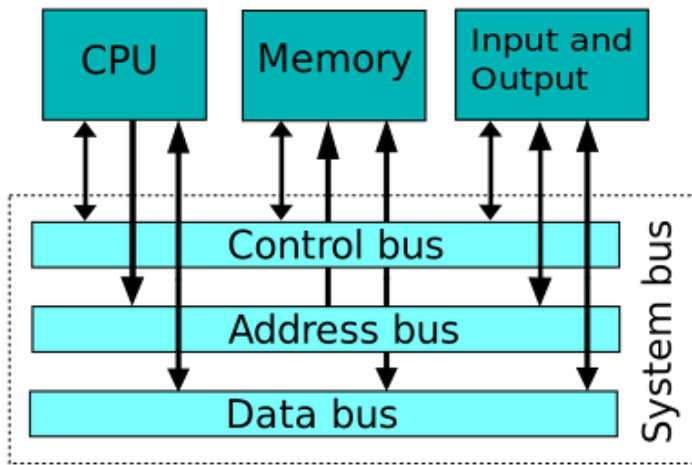


BUS

하드웨어 1



- BUS
- 전기적 신호가 오고 가는 **통로**
- System bus, PCIe, USB





하드웨어 필수편

Storage, ROM, 메인보드

ROM (Read Only Memory)

하드웨어 2



- ROM 고정 기억 장치
- 컴퓨터를 구동하기 위한 기본적인 정보
- 부트로더, 바이오스, 펌웨어 등을 저장하는 용도



Storage

하드웨어 2



- Storage 저장장치
- 데이터의 반영구 **저장**을 위해 사용
- **대용량** 데이터를 “**저장**”
- Floppy Disk, HDD(Hard Disk Drive), SSD(Solid State Drive), CD(Compact Disc), DVD(Digital Versatile Disc), Flash Memory 등등등

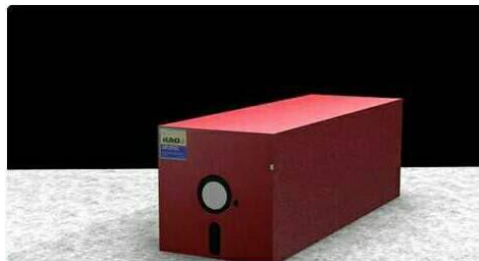
Goldenfir



SSD



HDD



Main board

하드웨어 2



- Main board 메인보드 마더보드
- 모든 하드웨어 디바이스들이 연결되는 보드
- CPU와 RAM 혹은 다른 장치들과 연결되는 버스, 전력공급, 인터페이스를 총괄







하드웨어 SIMD

그래픽(GPU) - SISD, SIMD

GPU (Graphics Processing Unit)

하드웨어 3



- GPU 그래픽 장치
- GPU와 RAM(그래픽용) 등등 추가한 보드를 그래픽 카드 라고 함
- 이미지를 연산해서 모니터에 출력
- Vector 연산, **다중 연산**이 동시다발적

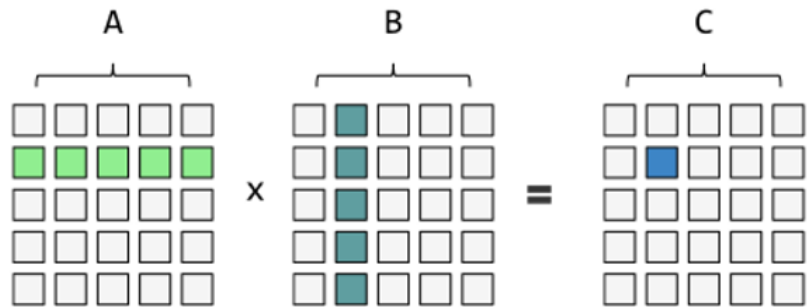
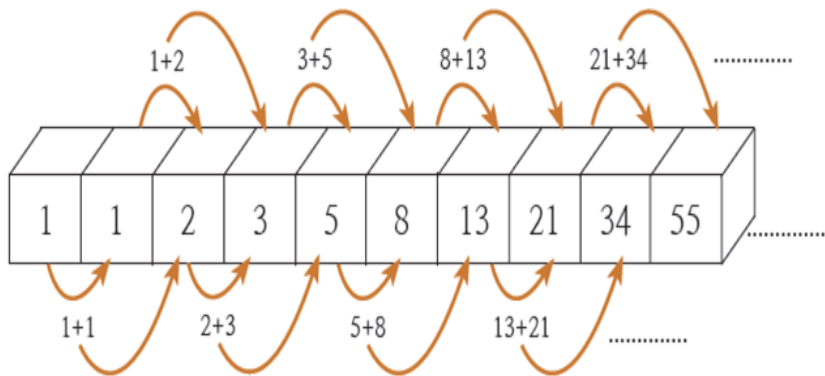


SISD, SIMD

하드웨어 3



- SISD(Single Instruction, Single Data)
- SIMD(Single Instruction, Multiple Data)
 - GPU, Intel의 MMX(MultiMedia eXtension), ARM의 NEON



$$C[i][j] = \text{sum}(A[i][k] * B[k][j]) \text{ for } k = 0 \dots n$$

GPU (Graphics Processing Unit)

하드웨어 3



- GPU의 역할
- 아주 많은 데이터를 동시에 연산
- 인해전술
- 빅데이터, 인공지능, 암호화폐 채굴, 게임, 3D

GPU 엔진 사양:

CUDA 코어	2560
베이스 클럭 (MHz)	1607
부스트 클럭 (MHz)	1733

프로그램 동작

C언어가 프로그램이 되는 과정





빌드와 실행

컴파일, 링크, 로드

Compile

프로그램의 동작



- Compile 컴파일
- User Friendly Language -> Processor Friendly Language
- C언어, 파이썬, 자바 -> 머신 코드(기계어)
- 처리장치들이 연산을 수행하기 위한 코드로 변경

Address	Machine Language				Assembly Language		
0000 0000	0000	0000	0000	0000	TOTAL	.BLOCK	1
0000 0001	0000	0000	0000	0010	ABC	.WORD	2
0000 0010	0000	0000	0000	0011	XYZ	.WORD	3
0000 0011	0001	1101	0000	0001	LOAD	REGD,	ABC
0000 0100	0001	1110	0000	0010	LOAD	REGE,	XYZ
0000 0101	0101	1111	1101	1110	ADD	REGF,	REGD, REGE
0000 0110	0010	1111	0000	0000	STORE	REGF,	TOTAL
0000 0111	1111	0000	0000	0000	HALT		

Compile

프로그램의 동작



```
int x = 10;
int y = 20;
int z = 30;
int m;

int func(int a)
{
    return a + 1;
}

void main()
{
    m = x + y + z;
    x = func(10);
    return;
}
```

Compiler



Code Section

0x0 func :
.....
0x10 main :
.....
.....

BSS Section

0x0 m :

Data Section

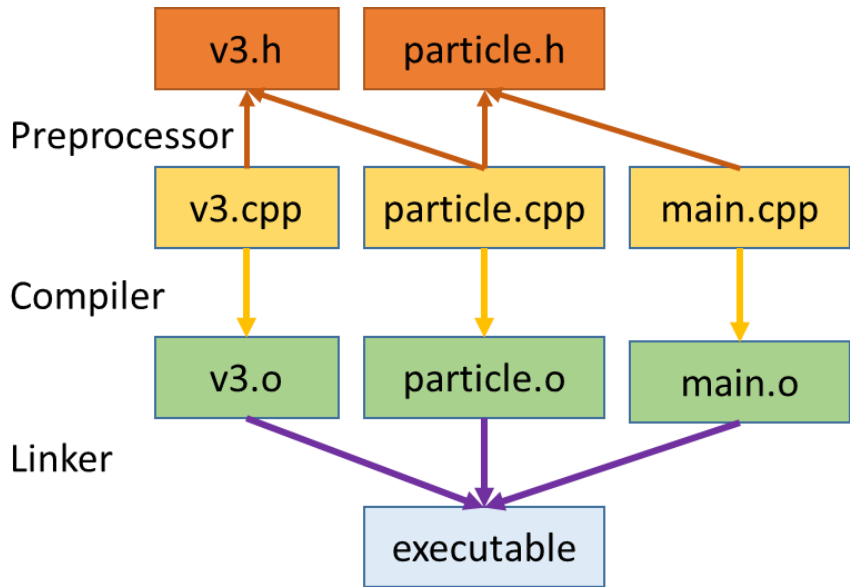
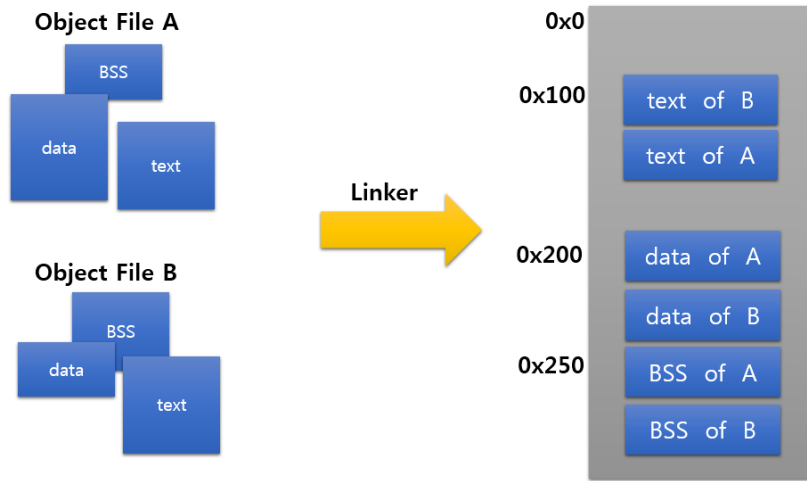
0x0 x : 10
0x4 y : 20
0x8 z : 30

Link

프로그램 동작



- Linking 링킹
- Undefined Symbol간의 연결
- Section 별로 재배치
- Executable File 생성

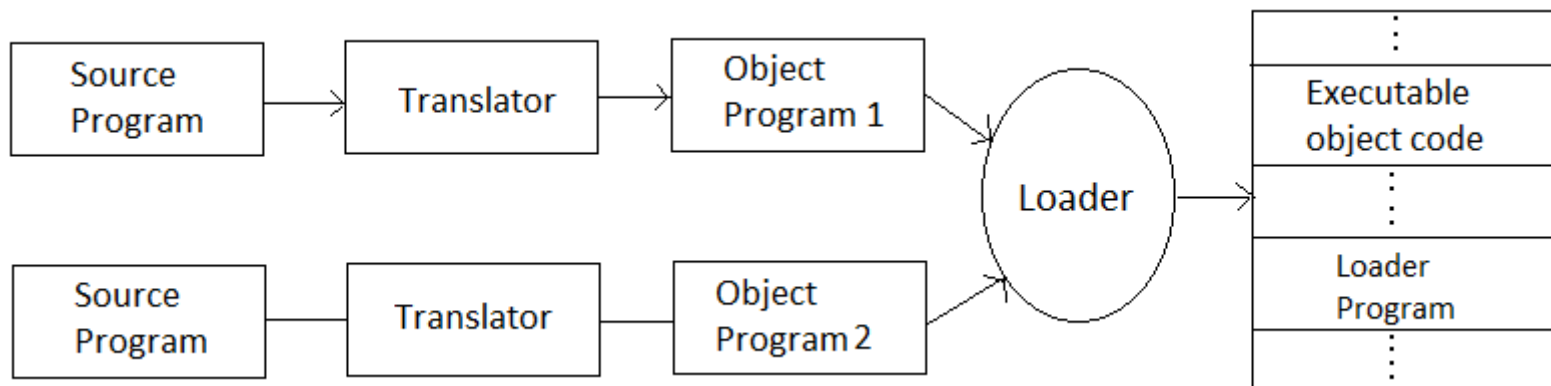


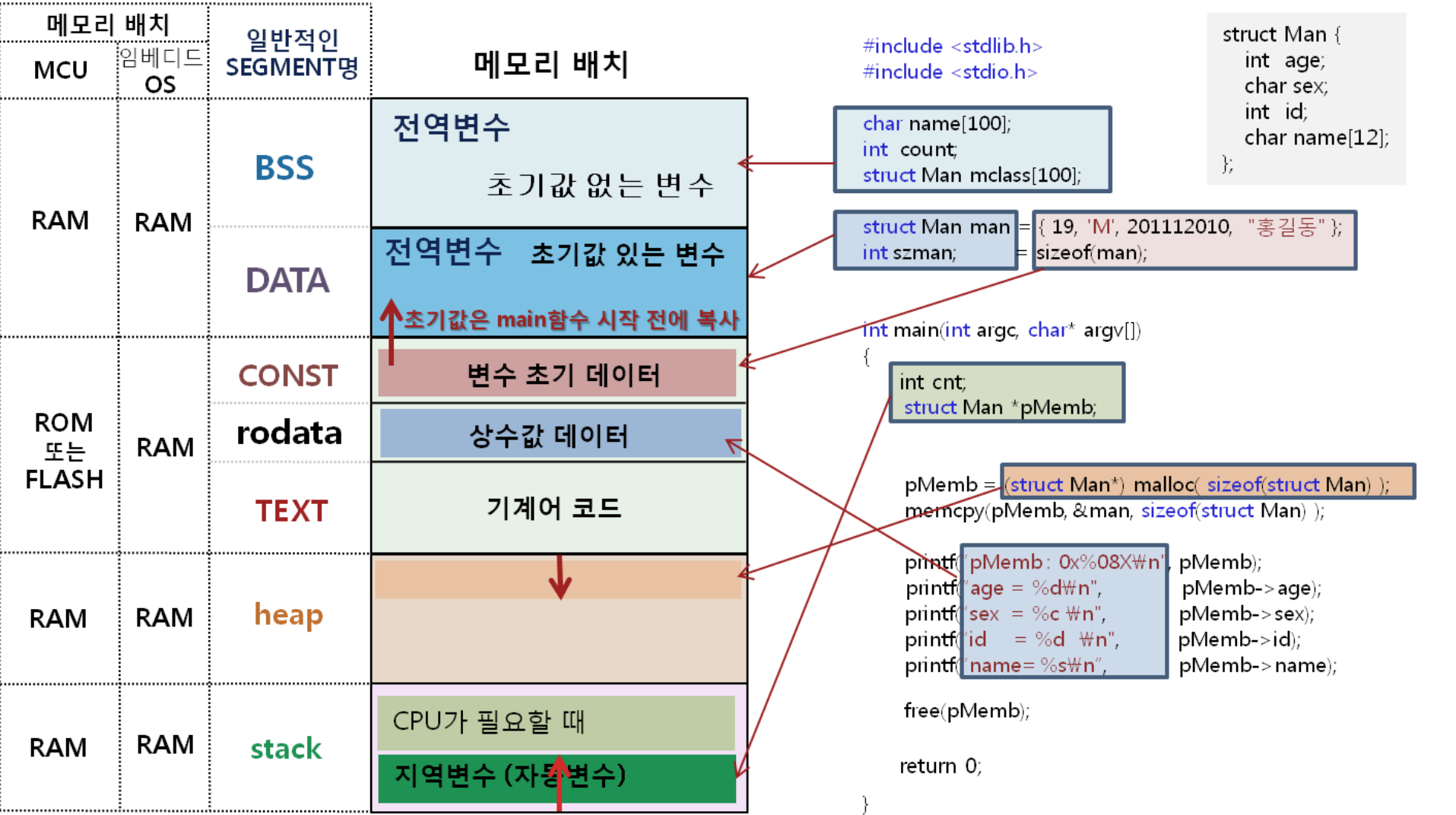
Load

프로그램 동작



- Loader 로더
- 프로그램을 실행시키면, 데이터들이 들어갈 공간을 메모리로부터 할당
- 할당 받은 메모리 공간에 데이터를 적재





Section	역할
Code	Machine code
Rodata	Constant, String literal
Data	Initialized global data space
BSS	Non-initialized global space
Heap	Dynamic allocated space
Shared Library	jaekodnae
Stack	Local data, Return address
Command Arguments	jaekodnaetoo
Kernel	kernel space

layout.c

2% 2/76 : 1

```

#include <stdio.h>
#include <stdlib.h>
int g_int;
static int gs_int;
int gi_int = 0xAA;
static int gsi_int = 0xBB;

int function(int);
int rec(unsigned long);

int main(int argc, char *argv[])
{
    auto int a_int;
    static int s_int;
    int ai_int = 0x11;
    static int si_int = 0x22;
    char *str_ltr = "hello";
    int *h_int = (int *)malloc(sizeof(int));
    int (*func)() = &function;

    func(1);

hear:
    printf("%16p: head:\n", &hear);

    return 0;
}

int function(int a)
{
    int i;
    printf("-----func-----\n");
    printf("%16p: parameter a\n", &a);
    printf("%16p: local i\n", &i);
    printf("%16p: return address(%16p)\n", &i +
5, (void *)*((long *)(&i + 5)));
    printf("-----\n");
    return 0;
}

```

N... layout.c

63% 48/76 : 7

tot0ro@T0T0Ro-Ub-Server:~/study/memory_layout\$./a.out

Code Section

0x56294983b72a: main

0x56294983b9a3: function

0x56294983b9a3: (*func)()

Rodata Section

0x56294983bb24: "hello"

0x56294983bb24: str_ltr

Data Section

0x562949a3d010: gi_int

0x562949a3d014: gsi_int

0x562949a3d018: si_int

BSS Section

0x562949a3d020: gs_int

0x562949a3d024: s_int

0x562949a3d028: g_int

Heap Section

0x56294a175260: h_int

Library Section

0x7fea20c4ee80: printf

Stack Section

0x7ffca8959d38: a_int

0x7ffca8959d3c: ai_int

Command Arguments

0x7ffca895c134: argv[0]: ./a.out

-----func-----

0x7ffca8959cfc: parameter a

0x7ffca8959d04: local i

0x7ffca8959d18: return address(0x56294983b970)

0x56294983b970: head:



머신코드

Feat. 최적화

x86 GCC -O0 옵션 시 생성 코드.

※ 후위 operand가 destination

test.c

```
1 int a;
2 int b = 2;
3 int main()
4 {
5     auto int c;
6     static int d;
7     static int e = 5;
8
9     a = b + e; // 7
10    c = a + e; // 12
11    d = c - b; // 10
12
13    if (a > d) {
14        d += d;
15    } else {
16        a += a;
17    }
18
19    while(1) {
20        if (a > 100)
21            break;
22        a++;
23    }
24
25    return 0;
26 }
27 }
```

nonoptimize.s

```
1 .file "test.c"
2 .text
3 .comm a,4,4
4 .globl b
5 .data
6 .align 4
7 .type b, @object
8 .size b, 4
9 b:
10 .long 2
11 .text
12 .globl main
13 .type main, @function
14 main:
15 .LFB0:
16 .cfi_startproc
17 pushq %rbp
18 .cfi_def_cfa_offset 16
19 .cfi_offset 6, -16
20 movq %rsp, %rbp
21 .cfi_def_cfa_register 6
22 movl b(%rip), %edx
23 movl e.1798(%rip), %eax
24 addl %edx, %eax
25 movl %eax, a(%rip)
26 movl a(%rip), %edx
27 movl e.1798(%rip), %eax
28 addl %edx, %eax
29 movl %eax, -4(%rbp)
30 movl b(%rip), %eax
31 movl -4(%rbp), %edx
32 subl %eax, %edx
33 movl %edx, %eax
34 movl %eax, d.1797(%rip)
35 movl a(%rip), %edx
36 movl d.1797(%rip), %eax
37 cmpl %eax, %edx
38 jle .L2
```

buffers

```
38 jle .L2
39 movl d.1797(%rip), %eax
40 addl %eax, %eax
41 movl %eax, d.1797(%rip)
42 jmp .L6
43 .L2:
44 movl a(%rip), %eax
45 addl %eax, %eax
46 movl %eax, a(%rip)
47 .L6:
48 movl a(%rip), %eax
49 cmpl $100, %eax
50 jg .L9
51 movl a(%rip), %eax
52 addl $1, %eax
53 movl %eax, a(%rip)
54 jmp .L6
55 .L9:
56 nop
57 movl $0, %eax
58 popq %rbp
59 .cfi_def_cfa 7, 8
60 ret
61 .cfi_endproc
62 .LFE0:
63 .size main, .-main
64 .data
65 .align 4
66 .type e.1798, @object
67 .size e.1798, 4
68 e.1798:
69 .long 5
70 .local d.1797
71 .comm d.1797,4,4
72 .ident "GCC: (Ubuntu 7.3.0-27ub
73 untu1-18.04) 7.3.0"
74 .section .note.GNU-stack,"",@prog
75 bits
```

N... nonoptimize.s

36% 27/73 : 7

nonoptimize.s

100% 73/73 : 6

x86 GCC -O2 옵션 시 생성 코드.

test.c

```
1 int a;
2 int b = 2;
3 int main()
4 {
5     auto int c;
6     static int d;
7     static int e = 5;
8
9     a = b + e; // 7
10    c = a + e; // 12
11    d = c - b; // 10
12
13    if (a > d) {
14        d += d;
15    } else {
16        a += a;
17    }
18
19    while(1) {
20        if (a > 100)
21            break;
22        a++;
23    }
24
25
26    return 0;
27 }
```

optimize.s

```
1 .file "test.c"
2 .text
3 .section .text.startup,"ax",@progbits
4 .p2align 4,,15
5 .globl main
6 .type main, @function
7 main:
8 .LFB0:
9     .cfi_startproc
10    movl b(%rip), %eax
11    addl $5, %eax
12    leal (%rax,%rax), %edx
13    cmpl $10, %eax
14    cmovle %edx, %eax
15    movl $101, %edx
16    cmpl $100, %eax
17    cmovle %edx, %eax
18    movl %eax, a(%rip)
19    xorl %eax, %eax
20    ret
21 .LFE0: .cfi_endproc
22 .size main, .-main
23 .globl b
24 .data
25 .align 4
26 .type b, @object
27 .size b, 4
28
29 b:
30 .long 2
31 .comm a,4,4
32 .ident "GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0"
33 .section .note.GNU-stack,"",@progbits
```

ELF dump

일반적인 SEGMENT명	메모리 배치
BSS	<div>전역변수</div> <div>초기값 없는 변수</div>
DATA	<div>전역변수 초기값 있는 변수</div> <div>초기값은 main 함수 시작 전에 복사</div>
CONST	변수 초기 데이터
rodata	상수값 데이터
TEXT	기계어 코드
heap	
stack	<div>CPU가 필요할 때</div> <div>지역변수 (자동변수)</div>

optimize.dump	444				285	4dc: 0f 1f 40 00	nopl	0x0(%rax)	buffers
	445	Disassembly of section .rodata:			286				
	446				287	Disassembly of section .plt.got:			
	447	00000000000006b0 <_IO_stdin_used>:			288				
	448	6b0: 01 00	add	%eax, (%rax)	289	00000000000004e0 <__cxa_finalize@plt>:			
	449	6b2: 02 00	add	(%rax), %al	290	4e0: ff 25 12 0b 20 00	jmpq	*0x200b12(%rip)	
	450					# 200ff8 <__cxa_finalize@GLIBC_2.2.5>			
	451	Disassembly of section .eh_frame_hdr:			291	4e6: 66 90	xchg	%ax, %ax	
optimize.dump	772	55% □ 449/816 □ : 30			292				
	773	Disassembly of section .data:			293	Disassembly of section .text:			
	774				294				
	775	000000000201000 <__data_start>:			295	00000000000004f0 <main>:			
	776	...			296	4f0: 8b 05 1a 0b 20 00	mov	0x200b1a(%rip), %eax	
	777					# 201010 			
	778	000000000201008 <__dso_handle>:			297	4f6: 83 c0 05	add	\$0x5, %eax	
	779	201008: 08 10	or	%dl, (%rax)	298	4f9: 8d 14 00	lea	(%rax, %rax, 1), %edx	
	780	20100a: 20 00	and	%al, (%rax)	299	4fc: 83 f8 0a	cmp	\$0xa, %eax	
	781	20100c: 00 00	add	%al, (%rax)	300	4ff: 0f 4e c2	cmovle	%edx, %eax	
	782	...			301	502: ba 65 00 00 00	mov	\$0x65, %edx	
	783				302	507: 83 f8 64	cmp	\$0x64, %eax	
	784	000000000201010 :			303	50a: 0f 4e c2	cmovle	%edx, %eax	
	785	201010: 02 00	add	(%rax), %al	304	50d: 89 05 05 0b 20 00	mov	%eax, 0x200b05(%rip)	
	786	...				# 201018 <__TMC_END__>			
	787				305	513: 31 c0	xor	%eax, %eax	
optimize.dump	788	94% □ 773/816 □ : 1			306	515: c3	retq		
	789	Disassembly of section .bss:			307	516: 66 2e 0f 1f 84 00 00	nopw	%cs: 0x0(%rax, %rax, 1)	
	790	000000000201014 <__bss_start>:			308	51d: 00 00 00			
	791	201014: 00 00	add	%al, (%rax)	309				
	792	...			310	0000000000000520 <_start>:			
	793				311	520: 31 ed	xor	%ebp, %ebp	
	794	000000000201018 <a>:			312	522: 49 89 d1	mov	%rdx, %r9	
	795	...			313	525: 5e	pop	%rsi	
	796				314	526: 48 89 e2	mov	%rsp, %rdx	
					315	529: 48 83 e4 f0	and	\$0xfffffffffffffffff0, %rs	
						p			
					316	52d: 50	push	%rax	
					317	52e: 54	push	%rsp	
					318	52f: 4c 8d 05 6a 01 00 00	lea	0x16a(%rip), %r8	@@@
optimize.dump		97% □ 793/816 □ : 1			N...	optimize.dump	35% □ 286/816 □ : 1		

test.c	nonoptimize.dump		nonoptimize.dump
1 int a;	374 5f4: 5d	pop %rbp	200fc0: 00 0e add %cl,(%rsi)
2 int b = 2;	375 5f5: e9 66 ff ff ff	jmpq 560 <register_tm_clones>	200fc2: 20 00 and %al,(%rax)
3 int main()	376		...
4 {	377 00000000000005fa <main>:		Disassembly of section .data:
5 auto int c;	378 5fa: 55	push %rbp	
6 static int d;	379 5fb: 48 89 e5	mov %rsp,%rbp	
7 static int e = 5;	380 5fe: 8b 15 0c 0a 20 00	mov 0x200a0c(%rip),%edx # 201010 	0000000000201000 <__data_start>:
8	381 604: 8b 05 0a 0a 20 00	mov 0x200a0a(%rip),%eax # 201014 <e.179>	...
9 a = b + e; // 7	382 60a: 01 d0	add %edx,%eax	
10 c = a + e; // 12	383 60c: 89 05 0e 0a 20 00	mov %eax,0x200a0e(%rip) # 201020 <a>	0000000000201008 <__dso_handle>:
11 d = c - b; // 10	384 612: 8b 15 08 0a 20 00	mov 0x200a08(%rip),%edx # 201020 <a>	201008: 08 10 or %dl,(%rax)
12	385 618: 8b 05 f6 09 20 00	mov 0x2009f6(%rip),%eax # 201014 <e.179>	20100a: 20 00 and %al,(%rax)
13 if (a > d) {	386 61e: 01 d0	add %edx,%eax	20100c: 00 00 add %al,(%rax)
14 d += d;	387 620: 89 45 fc	mov %eax,-0x4(%rbp)	...
15 } else {	388 623: 8b 05 e7 09 20 00	mov 0x2009e7(%rip),%eax # 201010 	
16 a += a;	389 629: 8b 55 fc	mov -0x4(%rbp),%edx	0000000000201010 :
17 }	390 62c: 29 c2	sub %eax,%edx	201010: 02 00 add (%rax),%al
18	391 62e: 89 d0	mov %edx,%eax	...
19 while(1) {	392 630: 89 05 e6 09 20 00	mov %eax,0x2009e6(%rip) # 20101c <d.179>	
20 if (a > 100)	393 636: 8b 15 e4 09 20 00	mov 0x2009e4(%rip),%edx # 201020 <a>	0000000000201014 <e.1798>:
21 break;	394 63c: 8b 05 da 09 20 00	mov 0x2009da(%rip),%eax # 20101c <d.179>	201014: 05 .byte 0x5
22 a++;	395 642: 39 c2	cmp %eax,%edx	201015: 00 00 add %al,(%rax)
23 }	396 644: 7e 10	jle 656 <main+0x5c>	...
24	397 646: 8b 05 d0 09 20 00	mov 0x2009d0(%rip),%eax # 20101c <d.179>	
25	398 64c: 01 c0	add %eax,%eax	Disassembly of section .bss:
26 return 0;	399 64e: 89 05 c8 09 20 00	mov %eax,0x2009c8(%rip) # 20101c <d.179>	
27 }	400 654: eb 0e	jmp 664 <main+0x6a>	0000000000201018 <__bss_start>:
~	401 656: 8b 05 c4 09 20 00	mov 0x2009c4(%rip),%eax # 201020 <a>	201018: 00 00 add %al,(%rax)
~	402 65c: 01 c0	add %eax,%eax	...
~	403 65e: 89 05 bc 09 20 00	mov %eax,0x2009bc(%rip) # 201020 <a>	
~	404 664: 8b 05 b6 09 20 00	mov 0x2009b6(%rip),%eax # 201020 <a>	000000000020101c <d.1797>:
~	405 66a: 83 f8 64	cmp \$0x64,%eax	20101c: 00 00 add %al,(%rax)
~	406 66d: 7f 11	jg 680 <main+0x86>	...
~	407 66f: 8b 05 ab 09 20 00	mov 0x2009ab(%rip),%eax # 201020 <a>	
~	408 675: 83 c0 01	add \$0x1,%eax	0000000000201020 <a>:
~	409 678: 05 a2 09 20 00	mov %eax,0x2009a2(%rip) # 201020 <a>	...
~	410 67e: eb e4	jmp 664 <main+0x6a>	
~	411 680: 90	nop	Disassembly of section .comment:
NORMAL test.c	NORMAL nonoptimize.dump	utf-8[unix] 35% 375/1056 : 46	NORMAL nonoptimize.dump utf-8[unix]

하드웨어 개념

하드웨어 부품들에 대한 개념과 이론

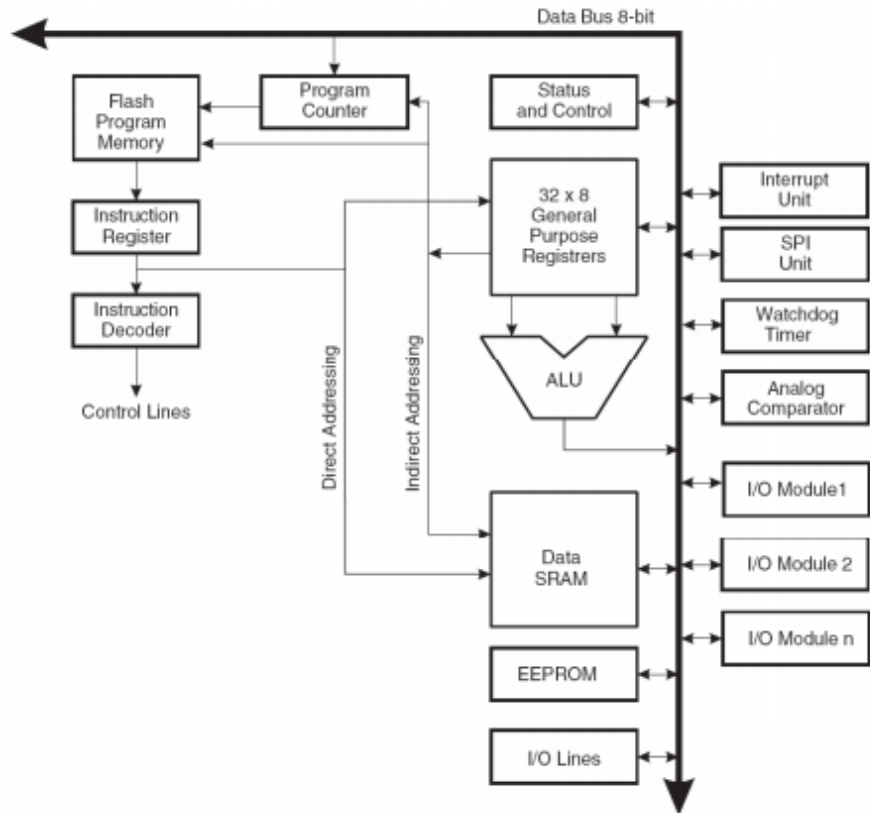


HW Components

하드웨어 개념



- 기본적인 데이터 흐름
- 최소한의 구조
- ATmega128





CPU

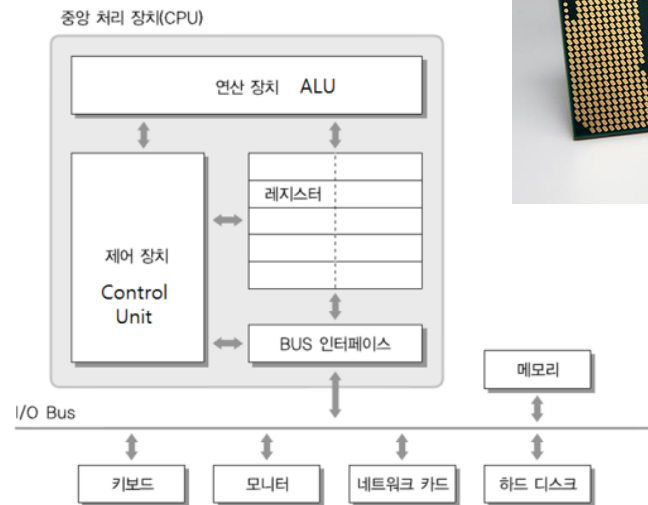
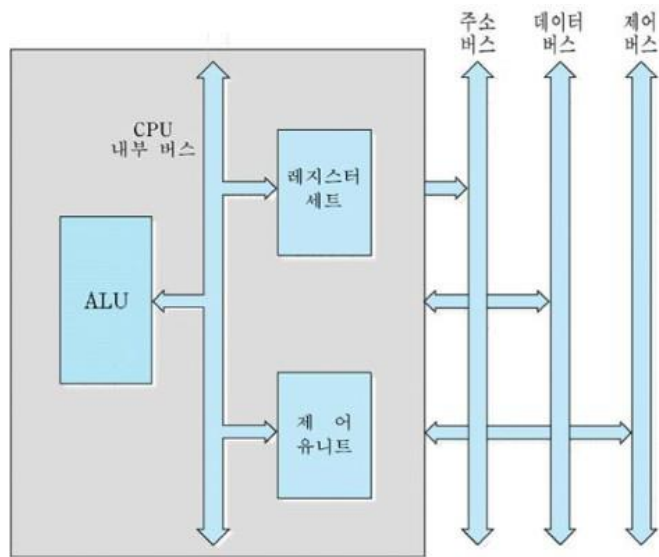
연산과 제어

역할과 구조

CPU



- 모든 처리와 연산, 제어를 담당



[그림 2-1] 80x86 시스템 구조

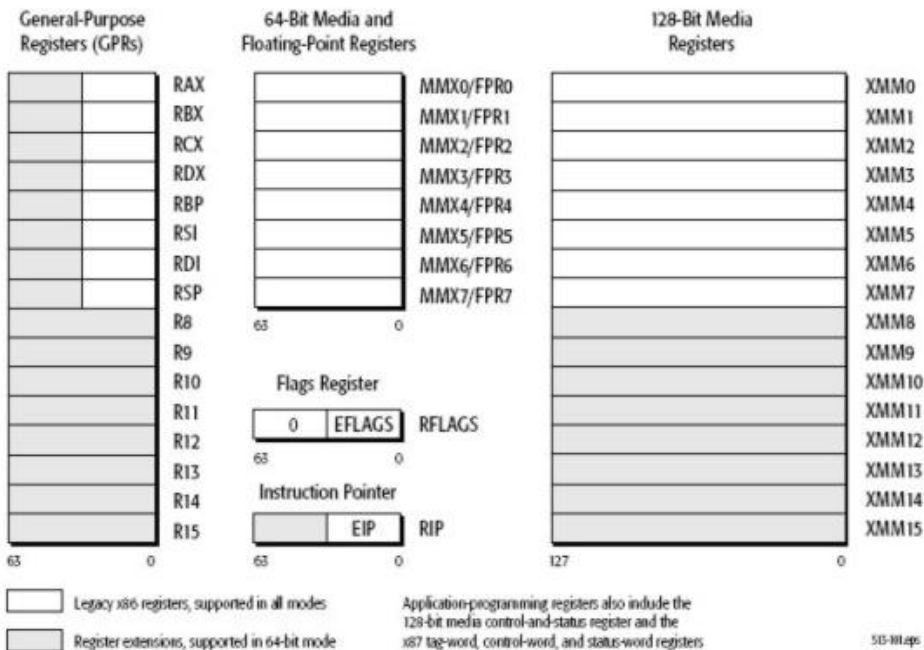


Register

CPU



- CPU가 당장 혹은 근시간에 연산에 사용하고 저장할 주소나 데이터



Register

CPU



- 메모리에서 데이터를 불러오는 것과 비교도 안 될 정도로 압도적 성능
- 물건이 필요할 때, 이미 들고 있는 것과 차고지에 다녀오는 수준의 차이

```
nonoptimize.dump
374 5f4: 5d                pop    %rbp
375 5f5: e9 66 ff ff ff      jmpq   560 <register_tm_clones>
376
377 00000000000005fa <main>:
378 5fa: 55                push   %rbp
379 5fb: 48 89 e5          mov     %rsp,%rbp
380 5fe: 8b 15 0c 0a 20 00  mov     0x200a0c(%rip),%edx    # 201010 <b>
381 604: 8b 05 0a 0a 20 00  mov     0x200a0a(%rip),%eax    # 201014 <e.1798>
382 60a: 01 d0            add     %edx,%eax
383 60c: 89 05 0e 0a 20 00  mov     %eax,0x200a0e(%rip)    # 201020 <a>
384 612: 8b 15 08 0a 20 00  mov     0x200ab8(%rip),%edx    # 201020 <a>
385 618: 8b 05 f6 09 20 00  mov     0x2009f6(%rip),%eax    # 201014 <e.1798>
386 61e: 01 d0            add     %edx,%eax
387 620: 89 45 fc          mov     %eax,-0x4(%rbp)
388 623: 8b 05 e7 09 20 00  mov     0x2009e7(%rip),%eax    # 201010 <b>
389 629: 8b 55 fc          mov     -0x4(%rbp),%edx
```

```
0000000000201010 <b>:
201010: 02 00                add     (%rax),%al
...

0000000000201014 <e.1798>:
201014: 05                .byte 0x5
201015: 00 00                add     %al,(%rax)
...

Disassembly of section .bss:

0000000000201018 <__bss_start>:
201018: 00 00                add     %al,(%rax)
...

000000000020101c <d.1797>:
20101c: 00 00                add     %al,(%rax)
...

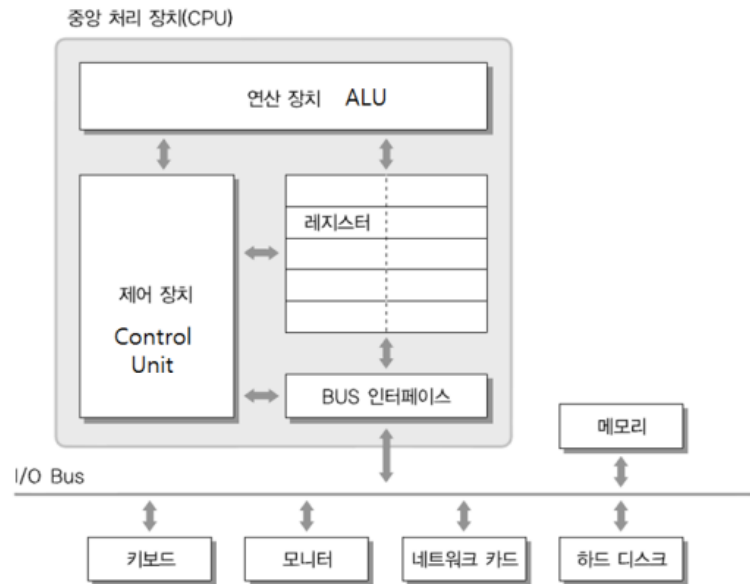
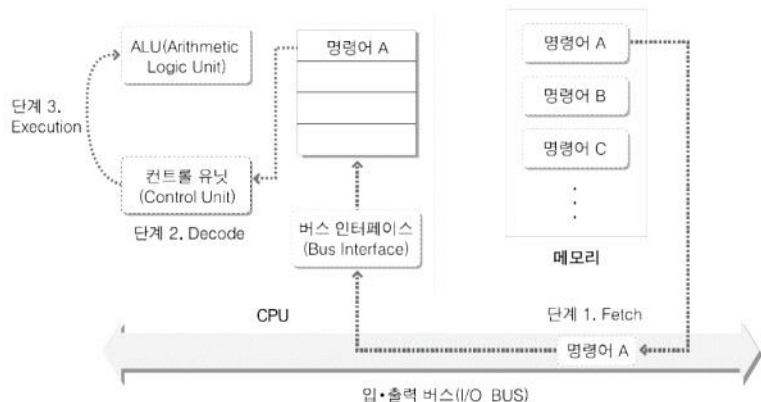
0000000000201020 <a>:
...
```

명령어 수행

CPU



- CPU의 basic한 명령어 수행과정
 1. 명령어를 인출
 2. 명령어를 해독. 레지스터 Read
 3. 명령어 수행. 연산 or 제어
 4. 메모리 접근. Load or Store
 5. 레지스터 Write Back



[그림 2-1] 80x86 시스템 구조

명령어 수행

CPU



- CPU의 basic한 명령어 수행과정
 1. 명령어를 인출
 2. 명령어를 해독. 레지스터 Read
 3. 명령어 수행. 연산 or 제어
 4. 메모리 접근. Load or Store
 5. 레지스터 Write Back

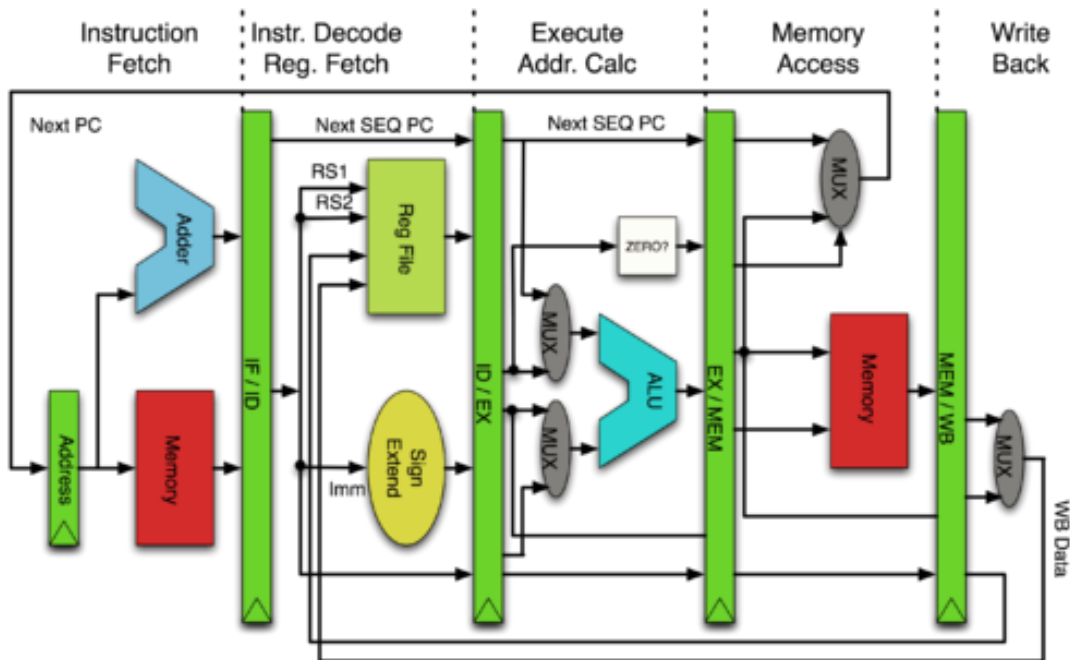
```
377 00000000000005fa <main>:
378 5fa: 55                push    %rbp
379 5fb: 48 89 e5          mov     %rsp,%rbp
380 5fe: 8b 15 0c 0a 20 00 mov     0x200a0c(%rip),%edx    # 201010 <b>
381 604: 8b 05 0a 0a 20 00 mov     0x200a0a(%rip),%eax    # 201014 <e.1798>
382 60a: 01 d0            add     %edx,%eax
383 60c: 89 05 0e 0a 20 00 mov     %eax,0x200a0e(%rip)    # 201020 <a>
384 612: 8b 15 08 0a 20 00 mov     0x200a08(%rip),%edx    # 201020 <a>
385 618: 8b 05 f6 09 20 00 mov     0x2009f6(%rip),%eax    # 201014 <e.1798>
386 61e: 01 d0            add     %edx,%eax
```

명령어 수행

CPU

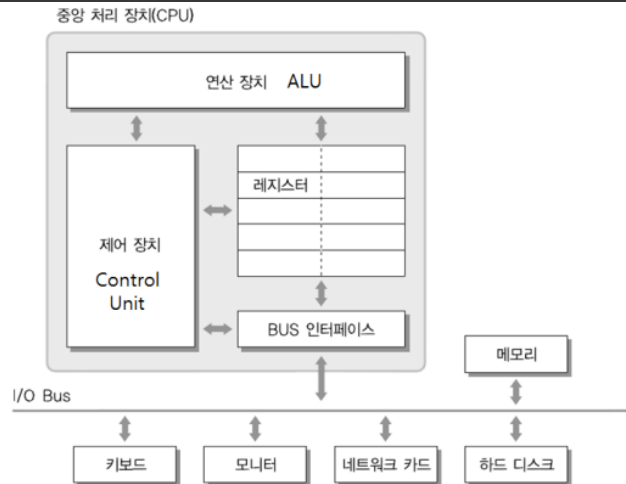


- CPU의 basic한 명령어 수행과정

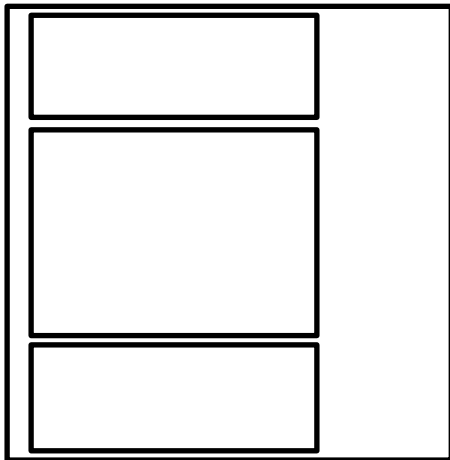


명령어 수행

CPU



[그림 2-1] 80x86 시스템 구조



```
push    %rbp
mov     %rsp,%rbp
mov     0x200a0c(%rip),%edx
mov     0x200a0a(%rip),%eax
add     %edx,%eax
mov     %eax,0x200a0e(%rip)
mov     0x200a08(%rip),%edx
mov     0x2009f6(%rip),%eax
add     %edx,%eax
```

377 00000000000005fa <main>:

```
378 5fa: 55                push    %rbp
379 5fb: 48 89 e5          mov     %rsp,%rbp
380 5fe: 8b 15 0c 0a 20 00 mov     0x200a0c(%rip),%edx    # 201010 <b>
381 604: 8b 05 0a 0a 20 00 mov     0x200a0a(%rip),%eax    # 201014 <e.1798>
382 60a: 01 d0            add     %edx,%eax
383 60c: 89 05 0e 0a 20 00 mov     %eax,0x200a0e(%rip)    # 201020 <a>
384 612: 8b 15 08 0a 20 00 mov     0x200a08(%rip),%edx    # 201020 <a>
385 618: 8b 05 f6 09 20 00 mov     0x2009f6(%rip),%eax    # 201014 <e.1798>
386 61e: 01 d0            add     %edx,%eax
```

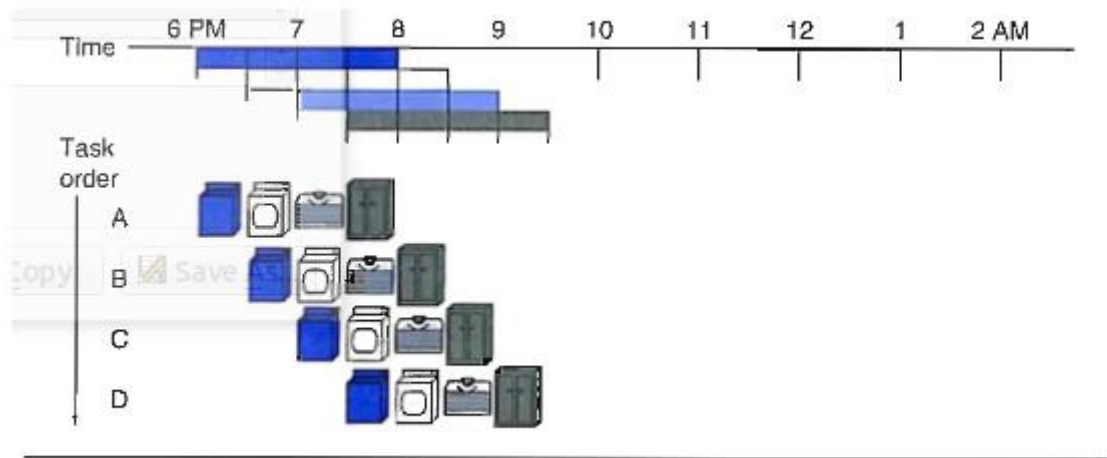
```
# 201010 <b>
# 201014 <e.1798>
# 201020 <a>
# 201020 <a>
# 201014 <e.1798>
```


Pipeline

CPU



- 파이프라인
- 자원을 쉬지 않게 함

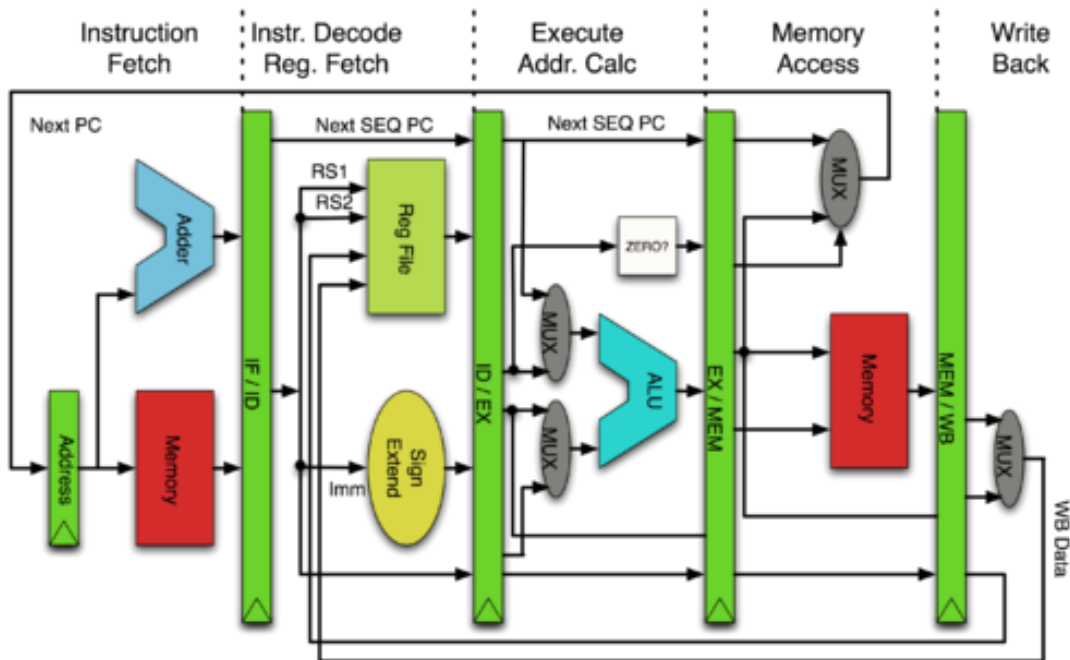


명령어 수행

CPU



- CPU의 basic한 명령어 수행과정

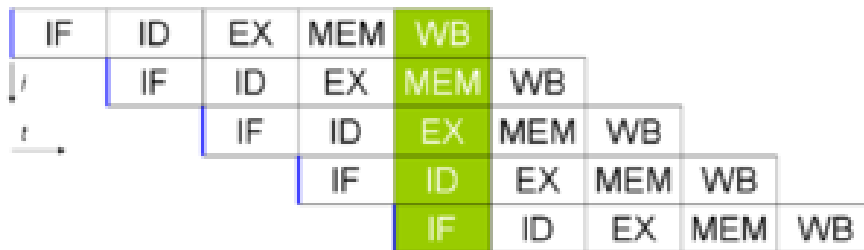


Pipeline

CPU



- 파이프라인
- 자원을 쉬지 않게 함

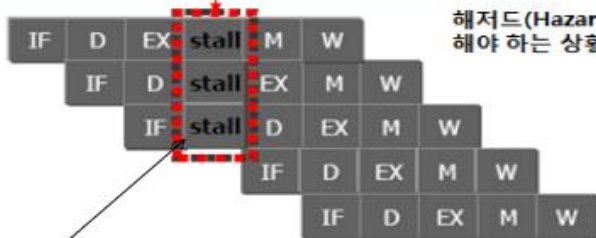


파이프라인 해저드

CPU



- 해저드 발생
 - 파이프라인을 정상적으로 수행하지 못하고 클럭을 낭비하게 되는 것
1. Structural Hazards 구조적 해저드
 2. Data Hazards 데이터 해저드
 3. Control Hazards 제어 해저드



해저드 (Hazard) : 파이프라인의 진행을 연기해야 하는 상황을 유발하는 원인

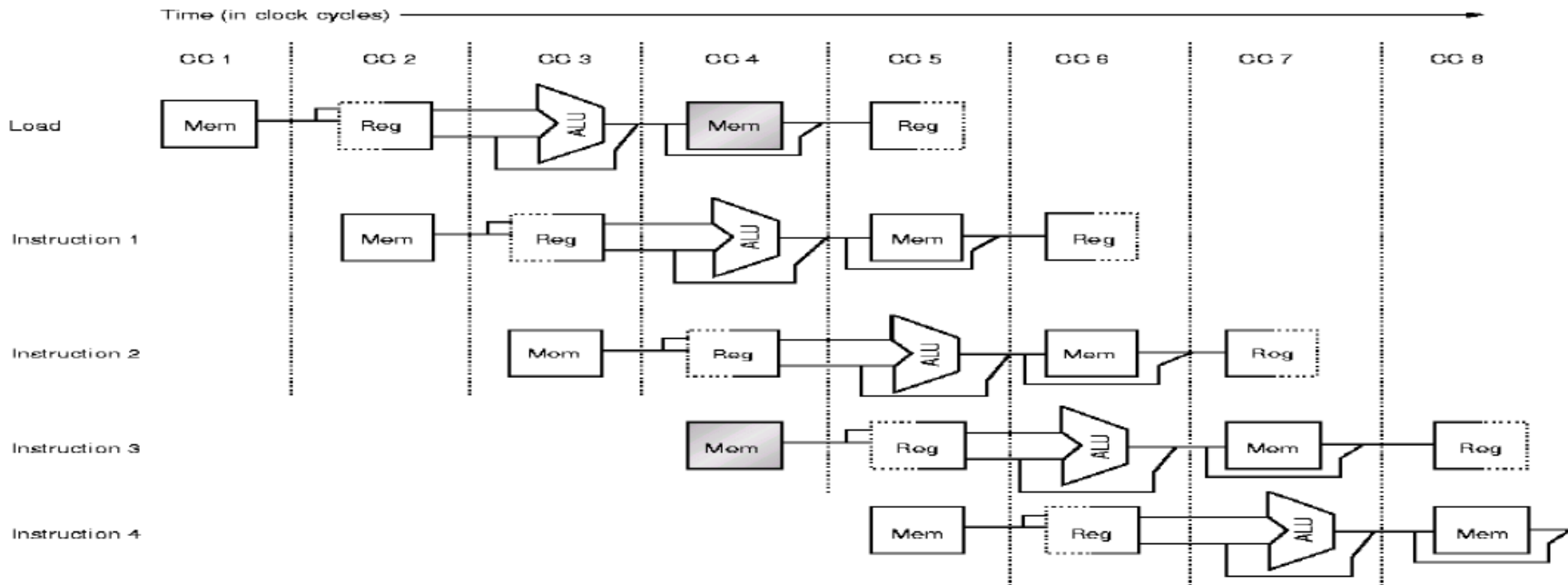
- Stall: 정지, 멎다 의미

Structural Hazards

CPU



- Structural Hazards 구조적 해저드
하드웨어가 여러 명령들의 수행을 지원하지 않기 때문에 발생, 자원충돌

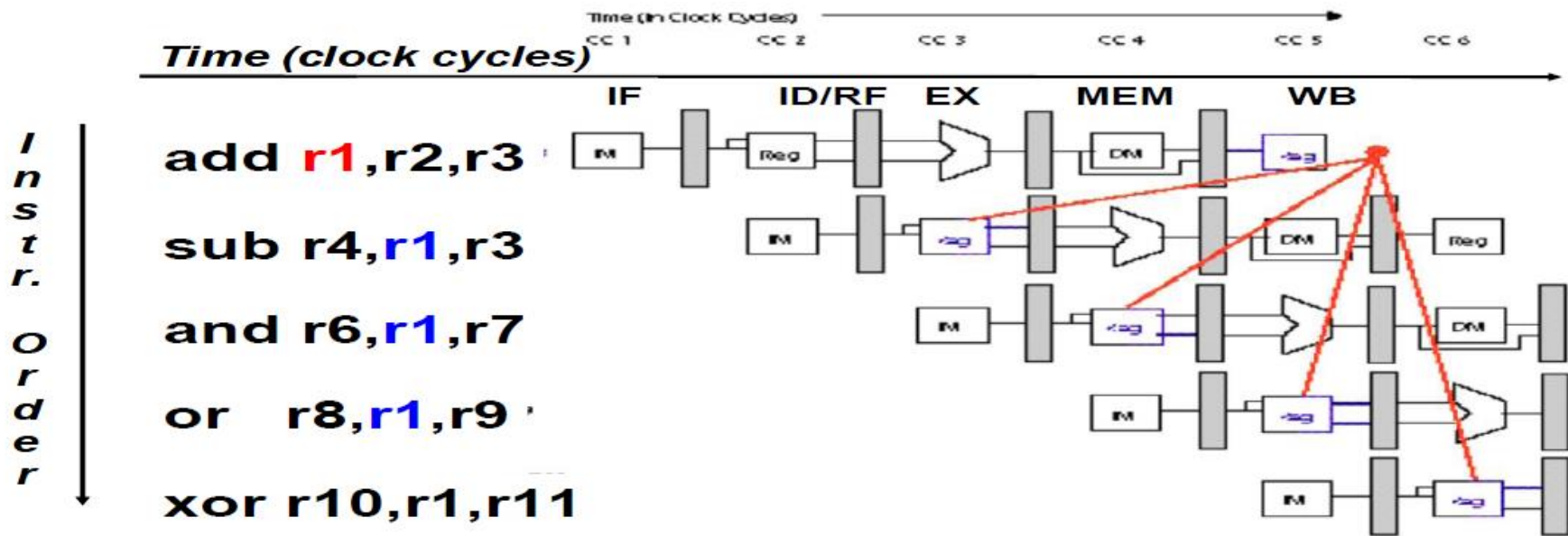


Data Hazards

CPU



- Data Hazards 데이터 해저드
명령의 값이 현재 파이프라인에서 수행 중인 이전 명령의 값에 종속 (세부적으로 RAW, WAR, WAW 해저드가 있음. RAR는 해저드 아님)



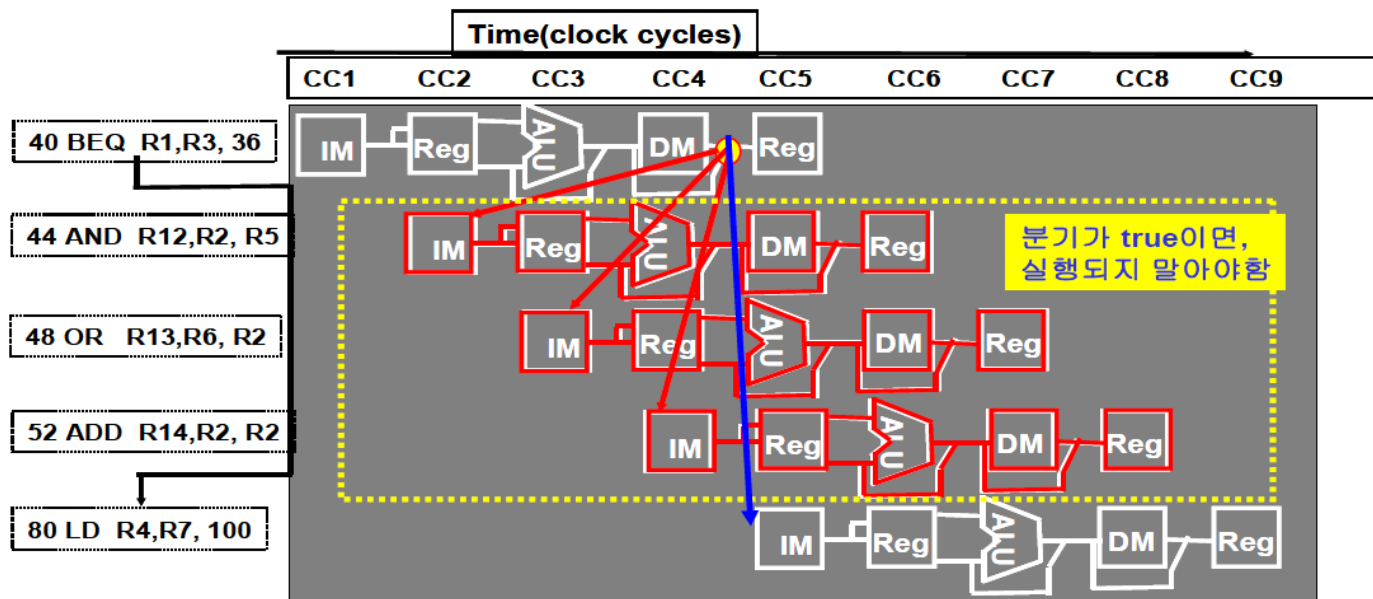
Control Hazards

CPU



- Control Hazards 제어 해저드

분기(jump, branch 등) 명령어에 의해서 발생 (분기를 결정된 시점에, 잘못된 명령이 파이프라인에 있기 때문에 발생)

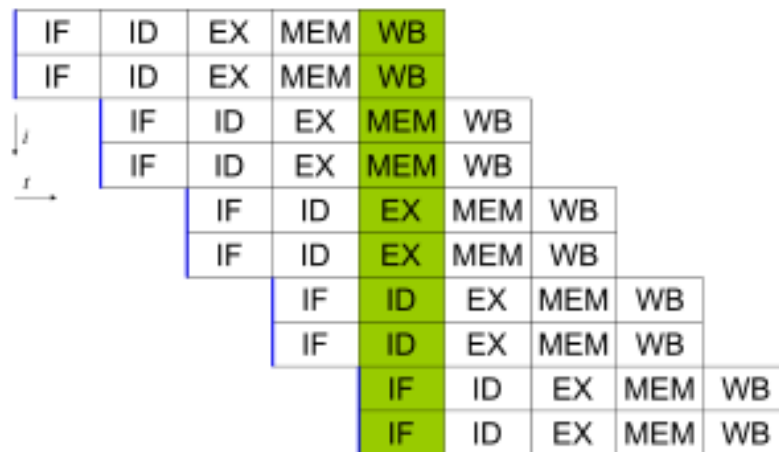
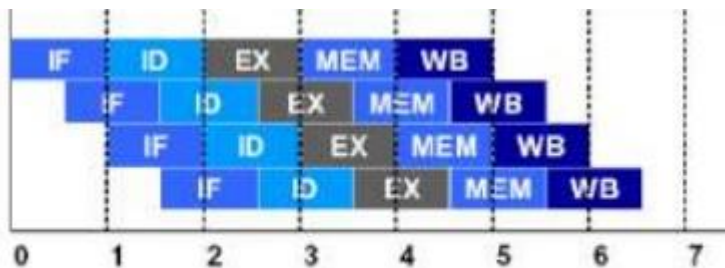


슈퍼파이프라인 & 슈퍼스칼라

CPU

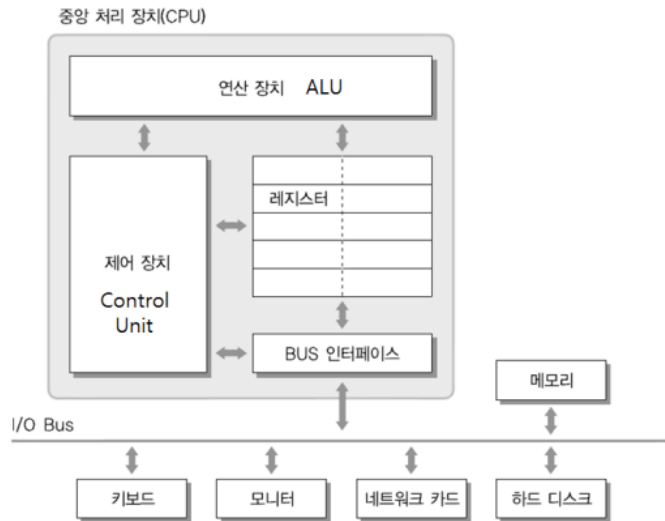


- 하나의 자원을 중복해서 사용함.

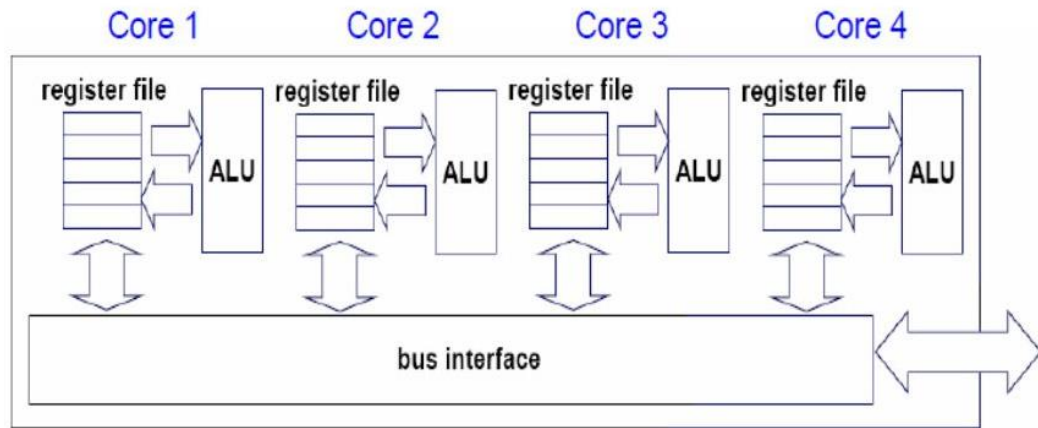


멀티코어

CPU



[그림 2-1] 80x86 시스템 구조



Multi-core CPU chip

엔디안

CPU



- 데이터를 바라보는 시각
사람 빅엔디안
원고지

2	3	4	5	6	7	8	9		
---	---	---	---	---	---	---	---	--	--

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

23	45	67	89		
----	----	----	----	--	--

0	1	2	3	4	5
---	---	---	---	---	---

2,345	6,789	
-------	-------	--

0	1	2
---	---	---

23,456,789	
------------	--

0

1

컴퓨터 리틀엔디안
메모리

9	8	7	6	5	4	3	2		
---	---	---	---	---	---	---	---	--	--

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

89	67	45	23		
----	----	----	----	--	--

0	1	2	3	4	5
---	---	---	---	---	---

6,789	2,345	
-------	-------	--

0	1	2
---	---	---

23,456,789	
------------	--


0

1

ISA (Instruction Set Architecture)

CPU

- ISA 명령어 집합
- Processing Unit이 처리할 수 있는 기계어 명령어 집합을 의미함
- ISA에 맞춰 프로세서를 개발
- 위에서 순서대로 AVR, ARM, X86



```
toto@TOTORo-Ub-Server: ~/assem/machinecode
optimize.dump | avropti.dump | armopti.dump | .vimrc |
00000028 <main>:
28: 80 91 60 00      lds    r24, 0x0060 ; 0x800060 <__data_start>
2c: 90 91 61 00      lds    r25, 0x0061 ; 0x800061 <__data_start+0x1>
30: 05 96            adiw   r24, 0x05 ; 5
32: 8b 30            cpi    r24, 0x0B ; 11
34: 91 05            cpc    r25, r1
36: a4 f0            brlt   .+40 ; 0x60 <__SREG__+0x21>
38: 90 93 63 00      sts    0x0063, r25 ; 0x800063 <__data_end+0x1>
3c: 80 93 62 00      sts    0x0062, r24 ; 0x800062 <__data_end>
40: 80 91 62 00      lds    r24, 0x0062 ; 0x800062 <__data_end>
44: 90 91 63 00      lds    r25, 0x0063 ; 0x800063 <__data_end+0x1>
48: 85 36            cpi    r24, 0x65 ; 101
4a: 91 05            cpc    r25, r1
avropti.dump
000000000000005f0 <main>:
5f0: b0000080      adrp   x0, 11000 <__data_start>
5f4: 90000081      adrp   x1, 10000 <__FRAME_END__+0xf830>
5f8: b9401000      ldr    w0, [x0, #16]
5fc: 11001405      add    w0, w0, #0x5
600: 7100281f      cmp    w0, #0xa
604: 5400004c      b.gt   60c <main+0x1c>
608: 531f7800      lsl    w0, w0, #1
60c: f947f422      ldr    x2, [x1, #4072]
610: f947f421      ldr    x1, [x1, #4072]
614: b9000040      str    w0, [x2]
618: b9400020      ldr    w0, [x1]
NORMAL | armopti.dump
000000000000004f0 <main>:
4f0: 8b 05 1a 0b 20 00      mov     0x200b1a(%rip),%eax      # 201010 <b>
4f6: 83 c0 05            add     $0x5,%eax
4f9: 8d 14 00            lea     (%rax,%rax,1),%edx
4fc: 83 f8 0a            cmp     $0xa,%eax
4ff: 0f 4e c2            cmovle  %edx,%eax
502: ba 65 00 00 00      mov     $0x65,%edx
507: 83 f8 64            cmp     $0x64,%eax
50a: 0f 4e c2            cmovle  %edx,%eax
50d: 89 05 05 0b 20 00      mov     %eax,0x200b05(%rip)      # 201018 <__TMC_END__>
513: 31 c0              xor     %eax,%eax
515: c3                retq
optimize.dump
```


CISC & RISC

CPU



- CISC (Complex Instruction Set Computer) – x86, x64 등
- RISC (Reduced Instruction Set Computer) – ARM, AVR 등

구분	CISC	RISC
CPU instruction	명령어 개수가 많고, 그 길이가 다양하며 실행사이클도 명령어 마다 다름	명령어 길이는 고정적이며, 워드와 데이터 버스크기가 모두 동일, 실행 사이클도 모두 동일
회로 구성	복잡	단순
메모리 사용	높은 밀도 메모리 사용이 효율적	낮은 밀도의 명령어 사용으로 메모리 사용이 비효율적
프로그램 측면	명령어를 적게 사용	상대적으로 많은 명령어가 필요, 파이프라인 사용
컴파일러	다양한 명령을 사용하므로 컴파일러가 복잡해짐	명령어 개수가 적어서 단순한 컴파일러 구현 가능
특성	하드웨어의 복잡	컴파일러의 복잡
클럭	복잡하고 낮은 클럭으로 동작	빠른 클럭으로 동작
CPU 제어	마이크로프로그램 제어 방식	하드와이어드 제어방식

CISC & RISC

CPU



- CISC (Complex Instruction Set Computer)
 - x86 – IA32 (Intel이 개발)
 - x64 – AMD64 (AMD가 개발)
 - 서버 컴퓨터, 범용 PC에서 사용
- RISC (Reduced Instruction Set Computer)
 - ARM – (ARM Holdings) 핸드폰 모바일, 라즈베리파이 등
 - AVR – (Atmel) 아두이노
 - MIPS – (MIPS Tech.) 플레이스테이션 2, PSP, 닌텐도64
 - PowerPC – (애플, IBM, 모토로라) Wii



RAM

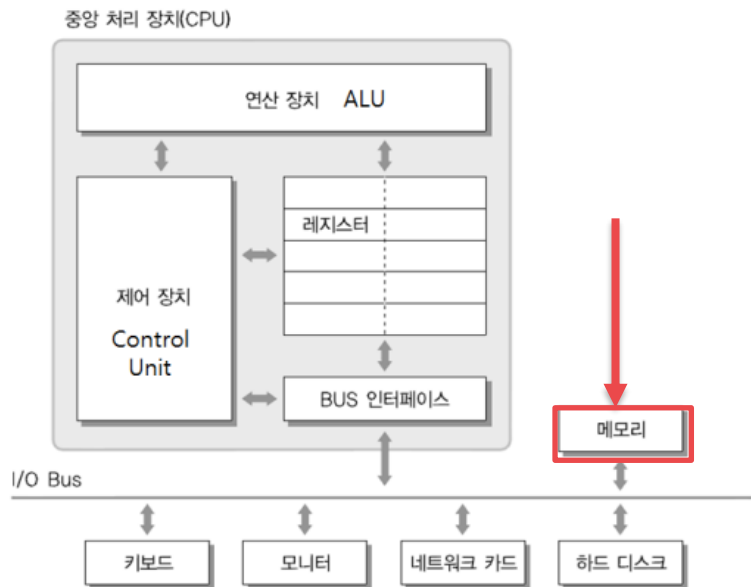
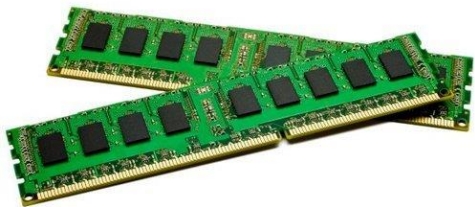
데이터 기억

역할

RAM



- RAM (Random Access Memory)
- 임의 접근 기억 장치
- 데이터가 저장된 위치에 상관 없이 데이터에 접근하는 속도가 일정
- 폰 노이만에 의해 등장
- CPU가 사용할 모든 데이터에 대해 기억하는 장소
- 프로그램이 동작하기 위해서 모든 Machine Code는 RAM에 존재해야함



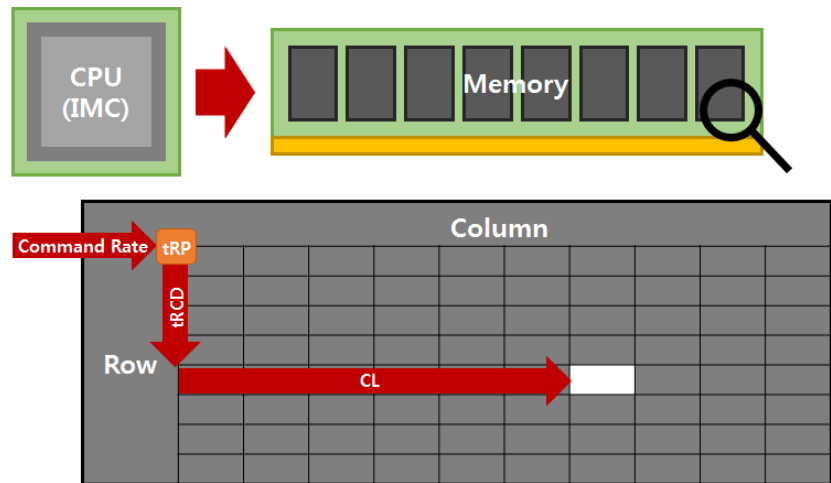
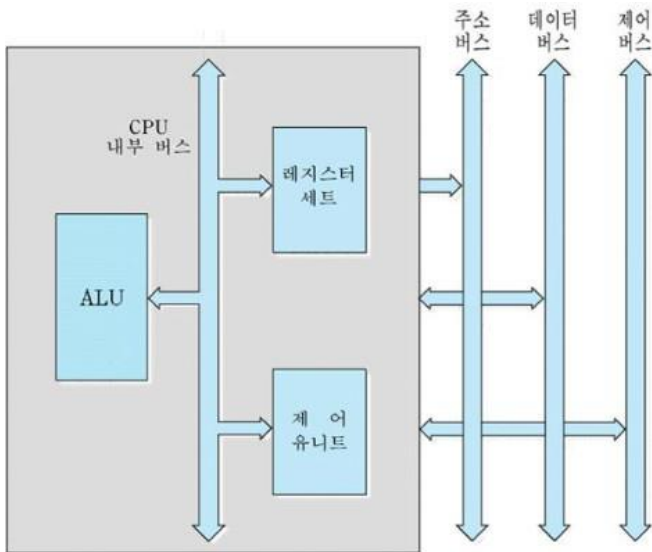
[그림 2-1] 80x86 시스템 구조

구조

RAM



- 워드 단위로 데이터에 접근
- Cell 하나당 1개의 주소를 가짐

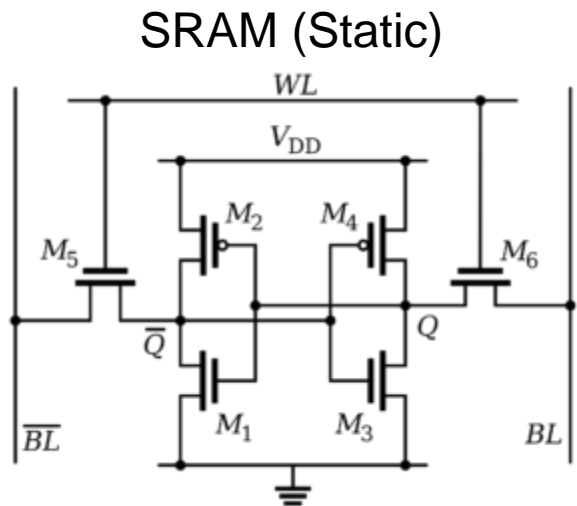
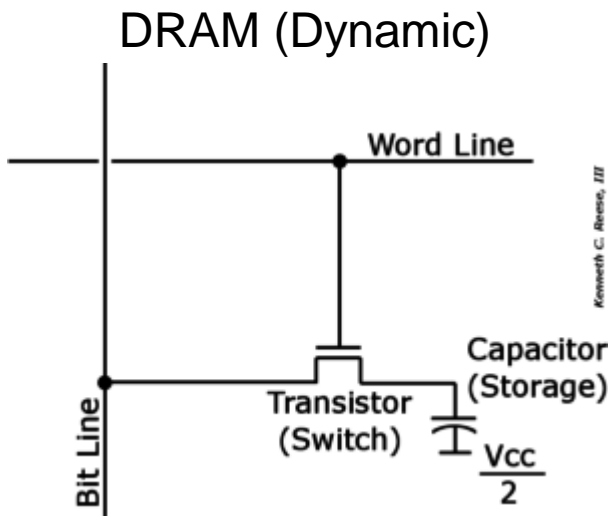


구조

RAM



- 워드 단위로 데이터에 접근
- Cell 하나당 1개의 주소를 가짐

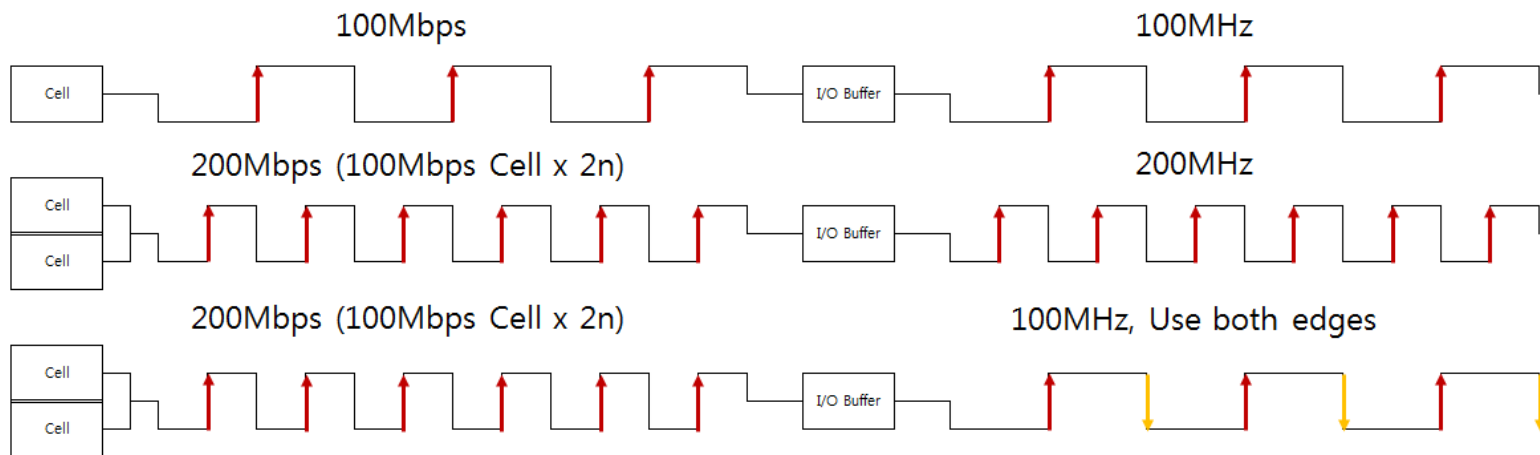


구조

RAM



- SDR(Single Data Rate)
- DDR(Double Data Rate)

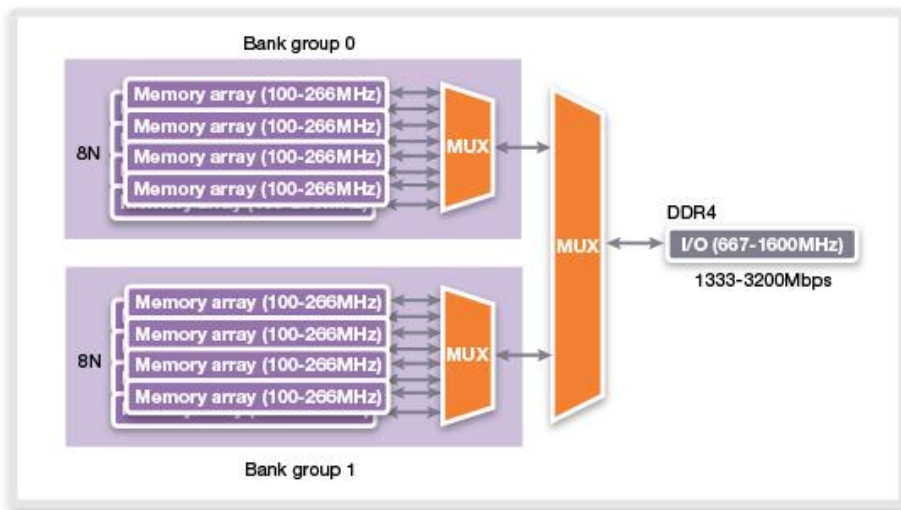
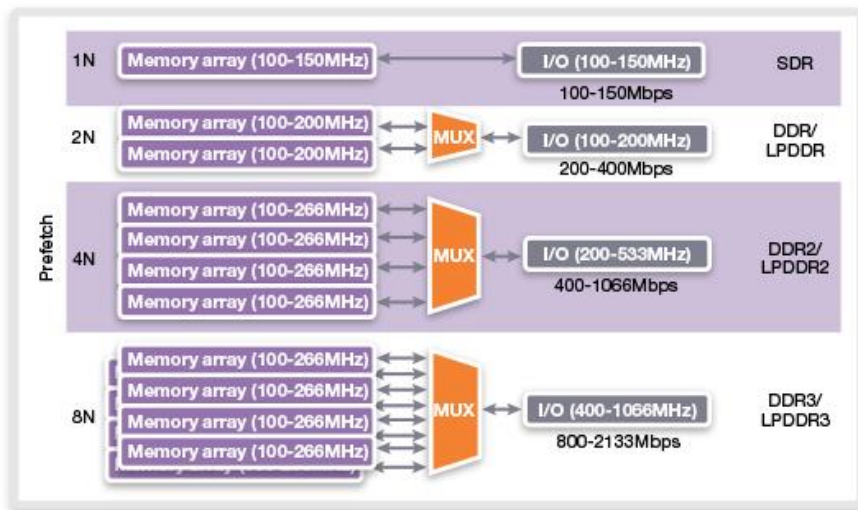


구조

RAM



- IO Buffer



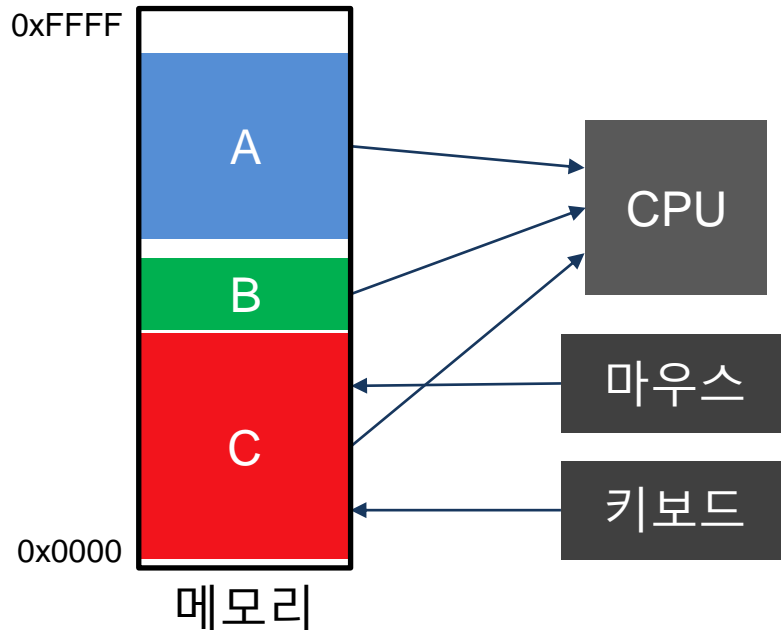
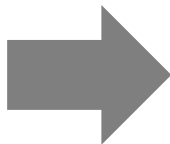
Physical Address

RAM



- 물리주소
- 실제 물리적인 메모리에 할당된 주소
- CPU나 하드웨어에서 메모리에 있는 데이터에 접근하기 위한 주소

실행하려는 프로그램



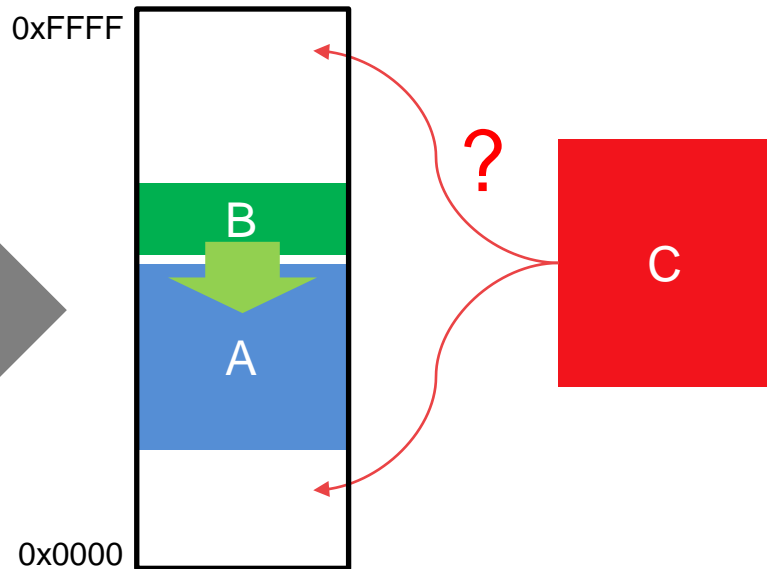
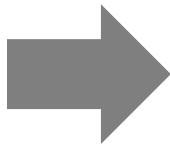
Physical Address

RAM



- 문제점
- Fragment 발생

실행하려는 프로그램



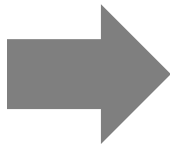
Page Frame

RAM

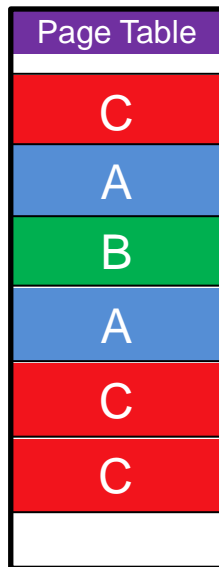


- 페이지 테이블
- 일정 크기로 관리단위를 만듦 => Page Frame
- 가상주소를 사용 => Page Table을 거쳐 물리주소로 변환

실행하려는 프로그램



0xFFFF



0x0000



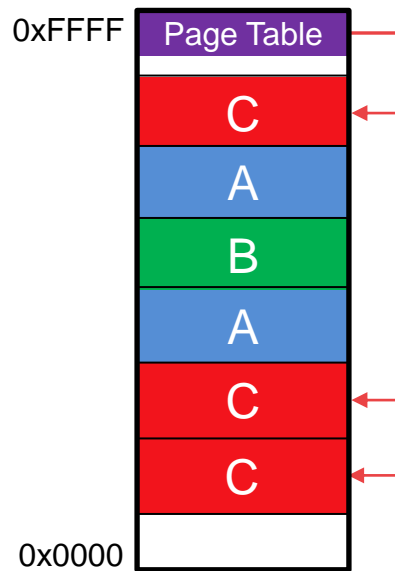
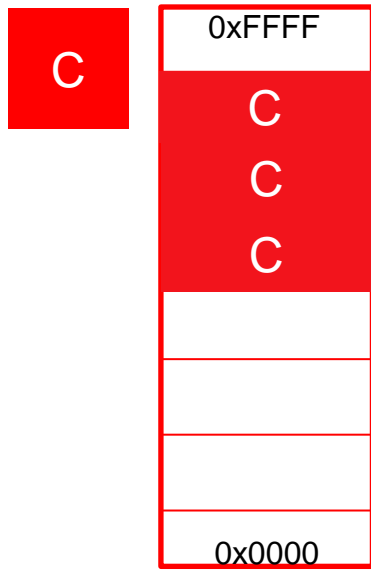
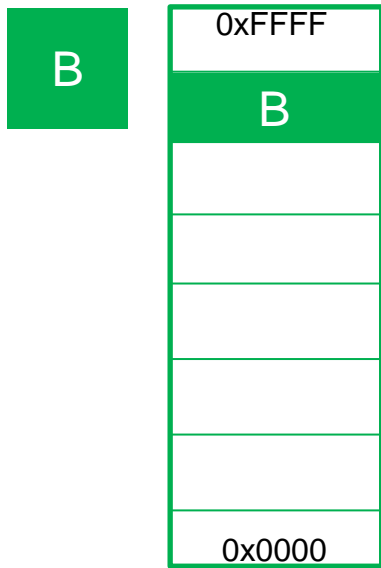
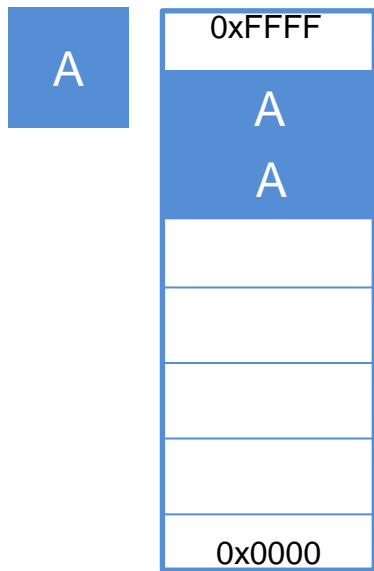
Virtual Address

RAM



- 가상주소
- 모든 동작하는 프로그램이 가상의 메모리를 가짐

실행하려는 프로그램



Virtual Address

RAM



- 가상주소
- 모든 동작하는 프로그램이 가상의 메모리를 가짐

실행하려는 프로그램

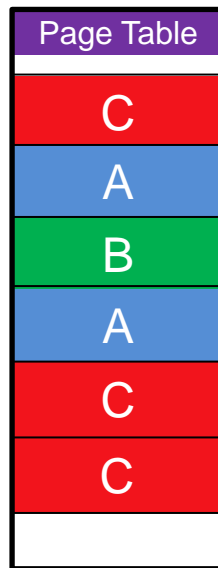


일반적인 SEGMENT명	메모리 배치
BSS	전역변수 초기값 없는 변수
DATA	전역변수 초기값 있는 변수 ↑ 초기값은 main 함수 시작 전에 복사
CONST	변수 초기 데이터
rodata	상수값 데이터
TEXT	기계어 코드
heap	↓
stack	CPU가 필요할 때 지역변수 (자동 변수)



일반적인 SEGMENT명	메모리 배치
BSS	전역변수 초기값 없는 변수
DATA	전역변수 초기값 있는 변수 ↑ 초기값은 main 함수 시작 전에 복사
CONST	변수 초기 데이터
rodata	상수값 데이터
TEXT	기계어 코드
heap	↓
stack	CPU가 필요할 때 지역변수 (자동 변수)

0xFFFF



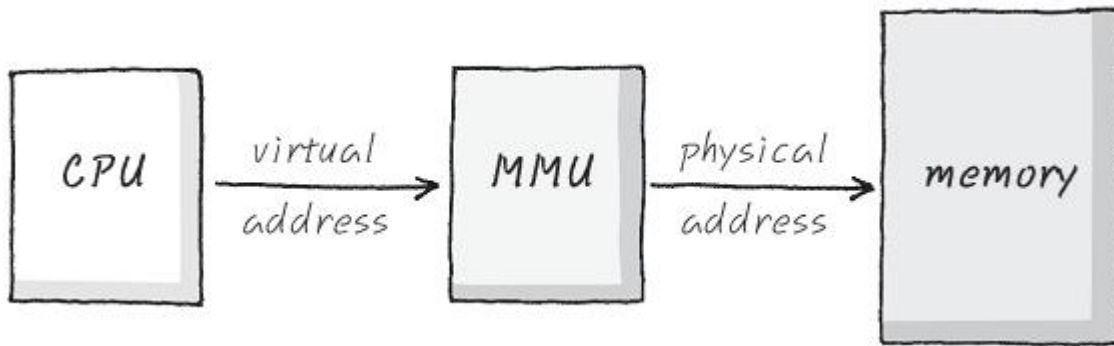
0x0000

MMU (Memory Management Unit)

RAM



- 메모리 관리 장치
- 가상주소를 물리주소로 변환하는 역할



MMU (Memory Management Unit)

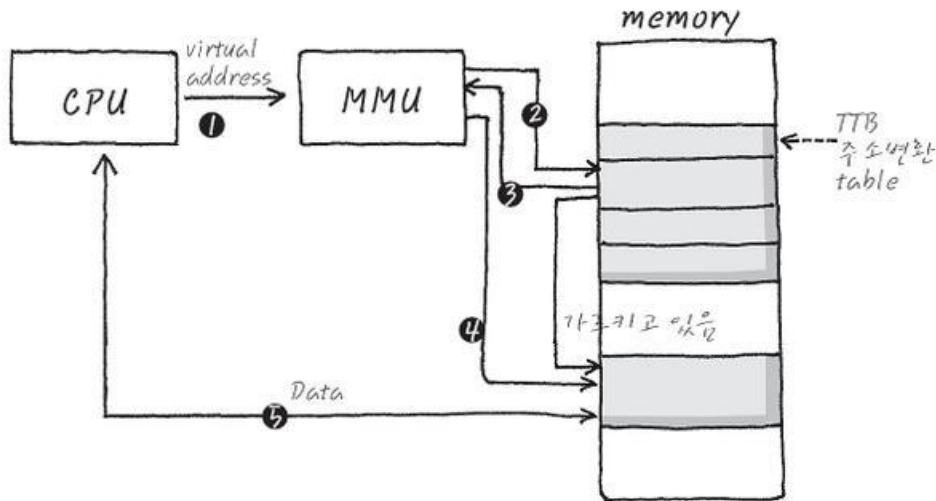
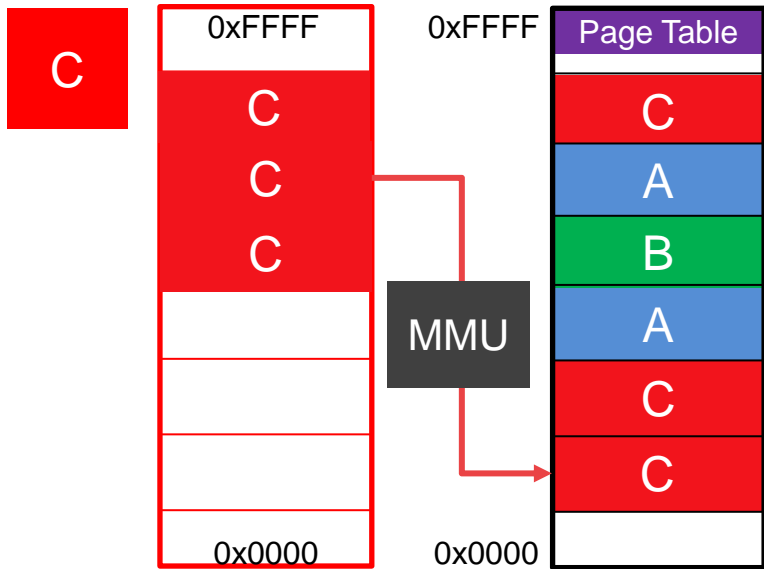
RAM



- 메모리 관리 장치
- 가상주소를 물리주소로 변환하는 역할

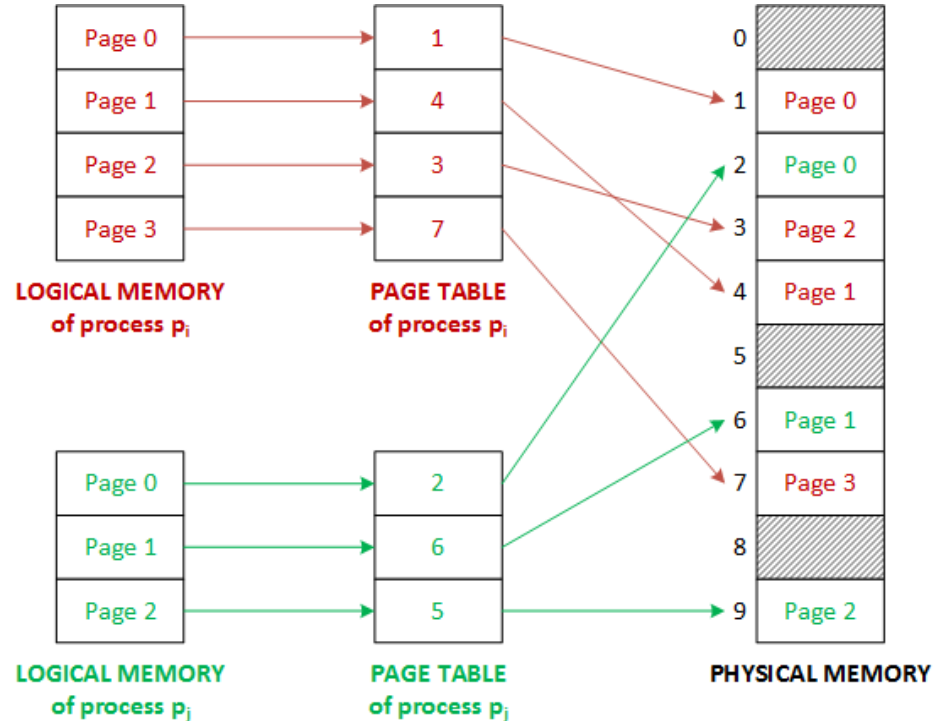
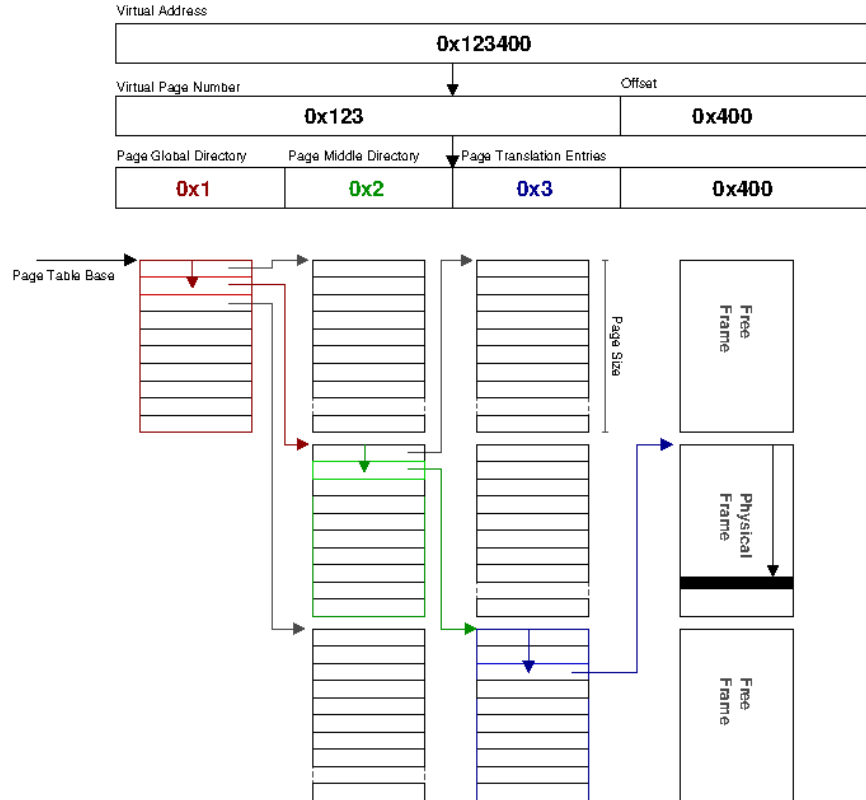
Cpu 사용 VM

실제 PM



MMU (Memory Management Unit)

RAM

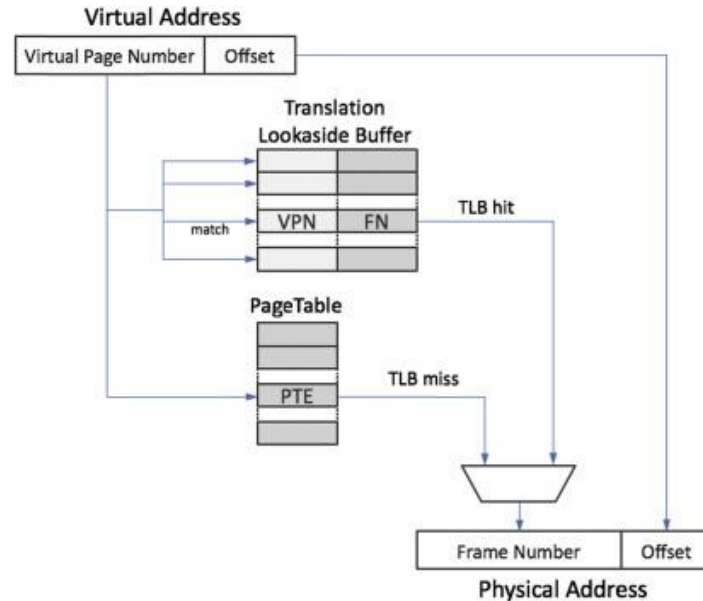


MMU (Memory Management Unit)

RAM

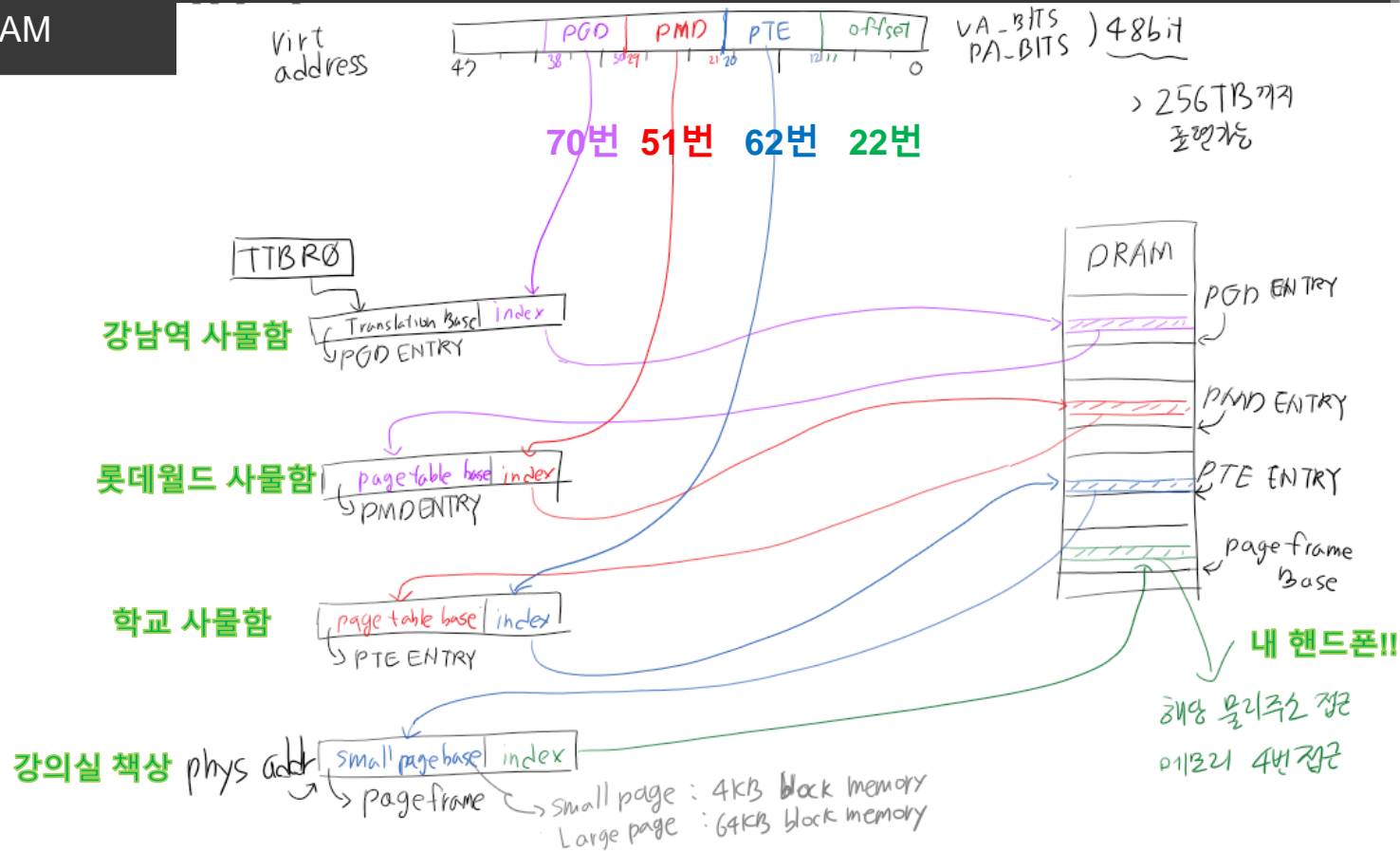


- Page Table
- TLB (Translation Lookaside Buffer)



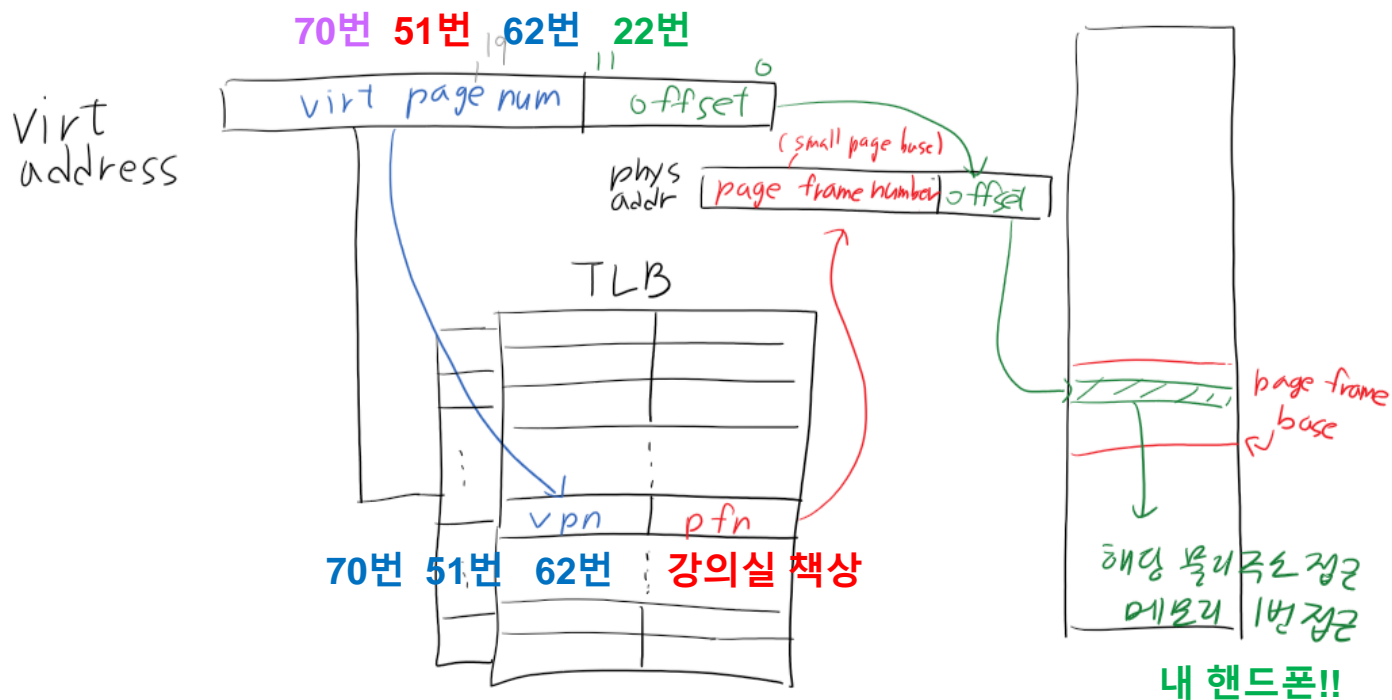
Virtual Address & Page Table

RAM



Virtual Address & TLB

RAM





CPU Cache

고속으로 처리하다.(feat. Google)

역할

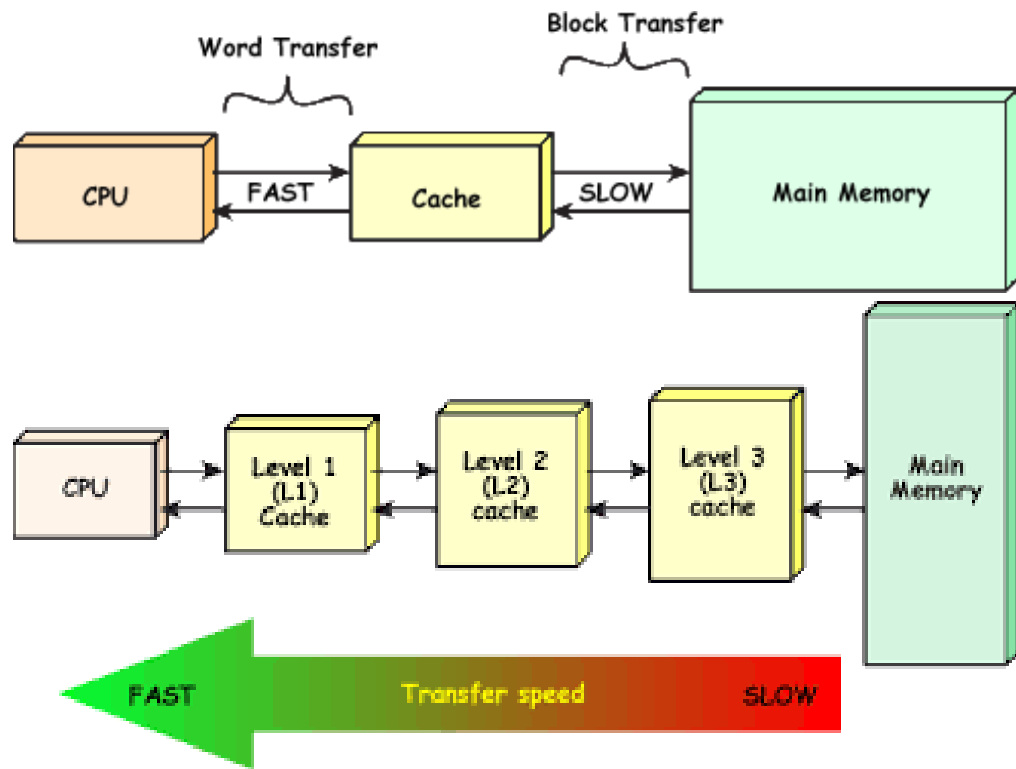
CPU Cache



- 메모리가 너무 느려서 탄생
- SRAM (Static Random Access Memory)
- 매우 작은 용량과 빠른 속도
- 메모리의 일부를 기억하고 있음
- Memory Locality의 원리

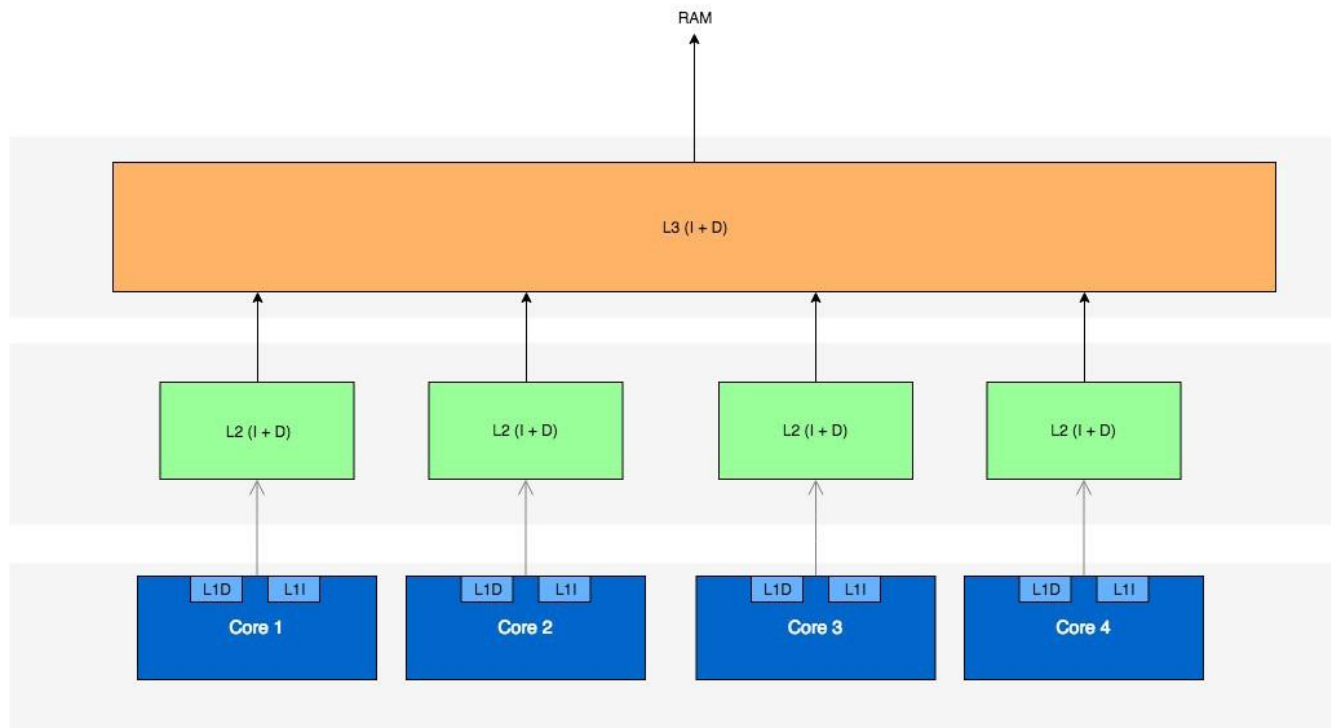
구조

CPU Cache



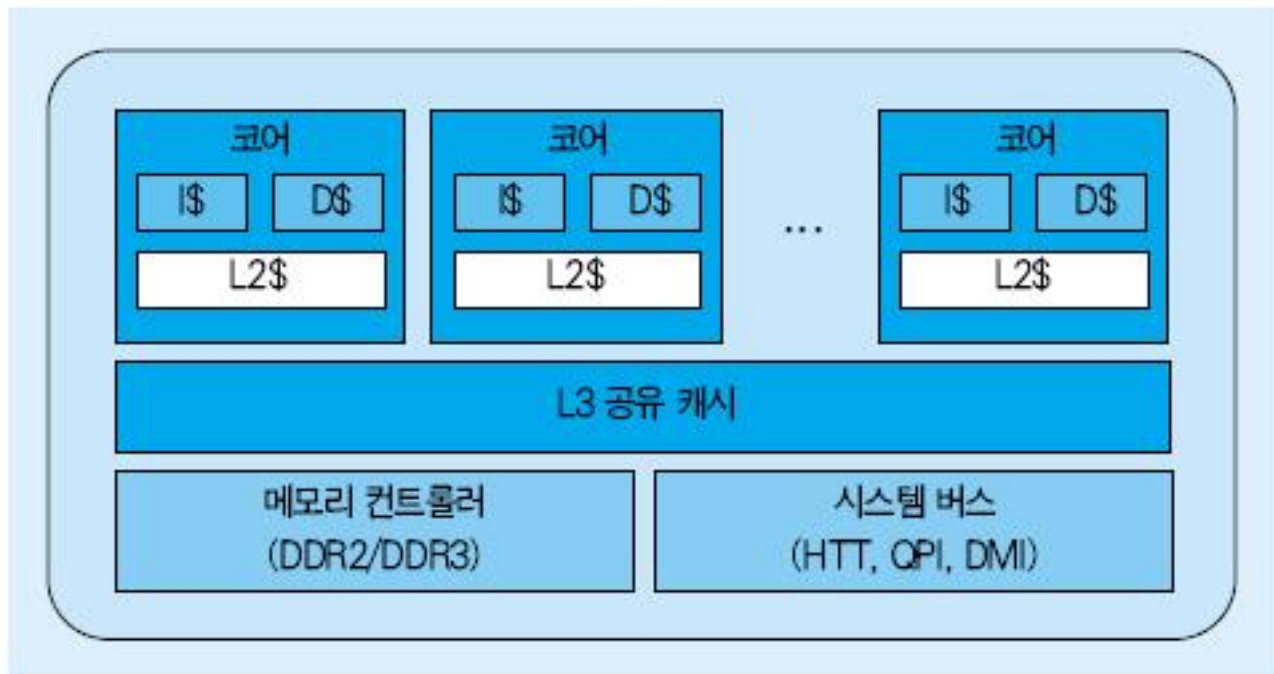
구조

CPU Cache



구조

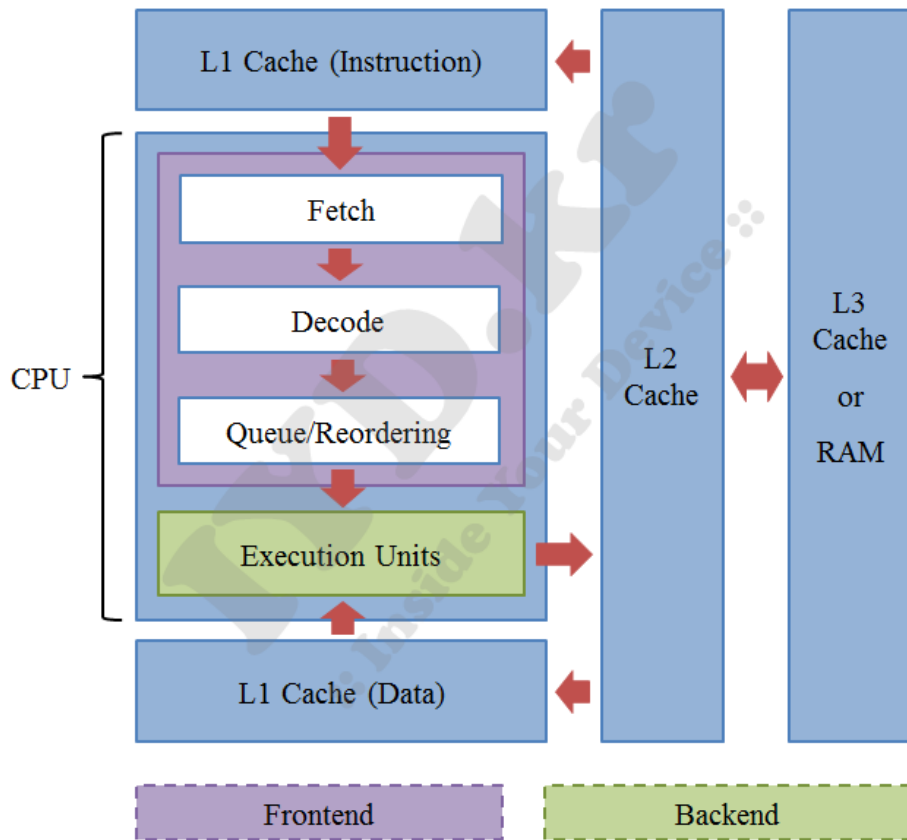
CPU Cache



〈그림 1〉 AMD/인텔의 최신 멀티 코어 프로세서 구조

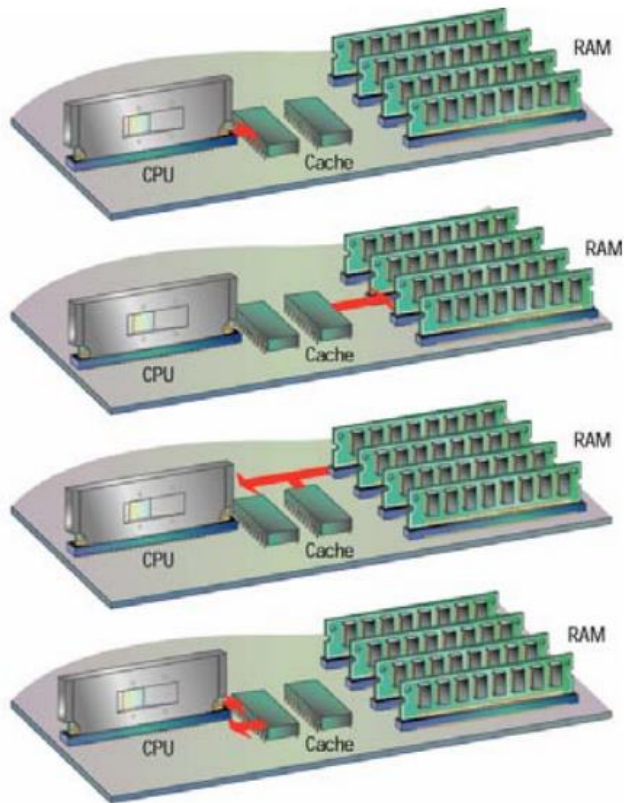
구조

CPU Cache



구조

CPU Cache



Memory Locality

CPU Cache

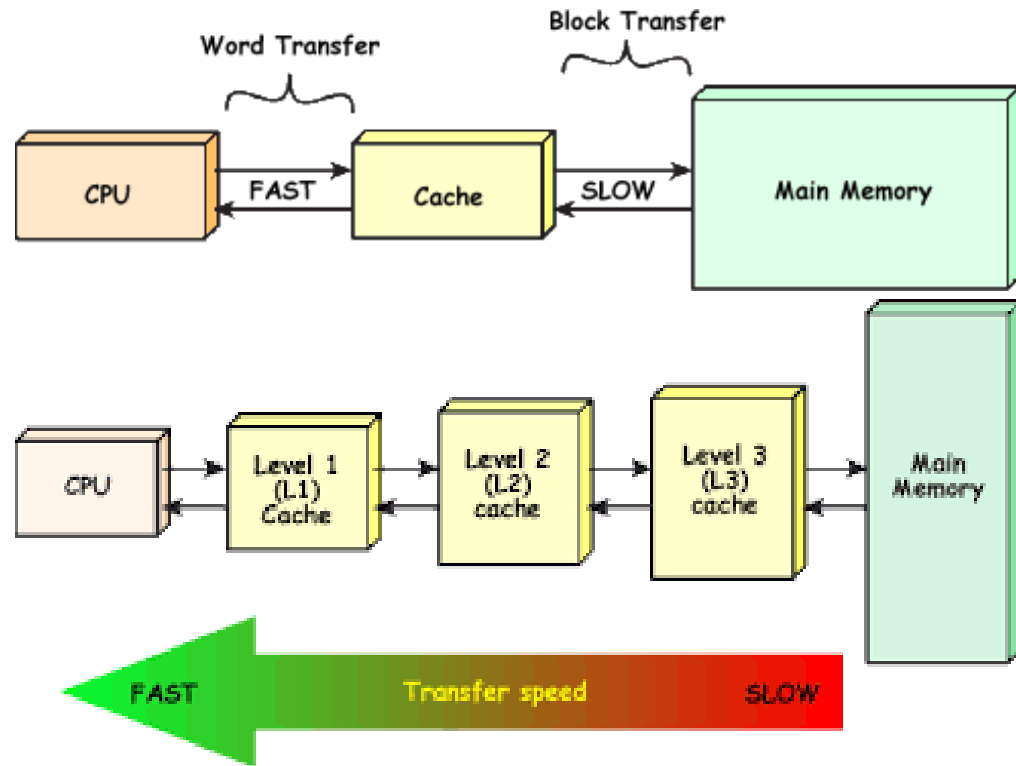


- 메모리 지역성
 1. 시간지역성 – 인접한 데이터가 이용되는 성질
 2. 공간지역성 – 데이터가 반복해서 이용되는 성질

```
void main ()  
{  
    int a;  
    int b;  
    int c;  
  
    a = 1;  
    b = 3;  
    c = 5;  
    a = a * b * c;  
    b = a - c + b;  
    c = a / b;  
    b = a * c;  
  
    a = 1;  
    b = 2;  
    c = 3;  
    while (b > 1000000) {  
        a += a;  
        b = a - b;  
        c = b + b;  
    }  
  
    return 0;  
}
```

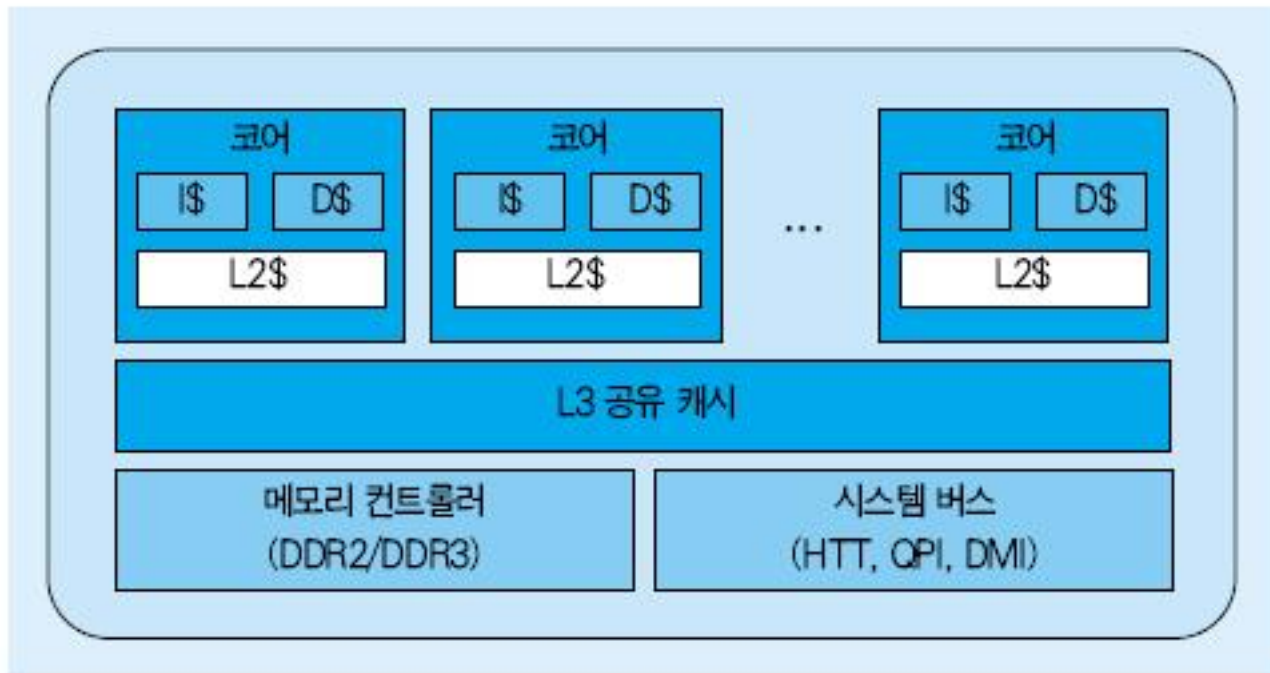
Memory hierarchy

CPU Cache



Memory hierarchy

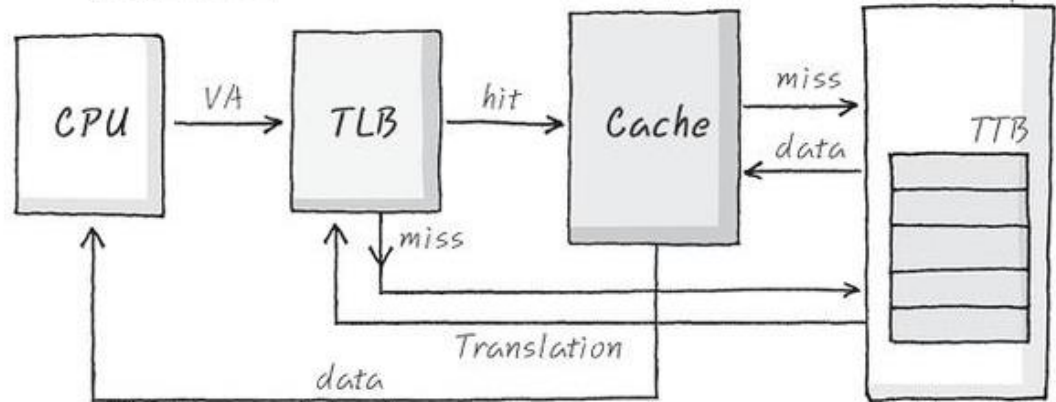
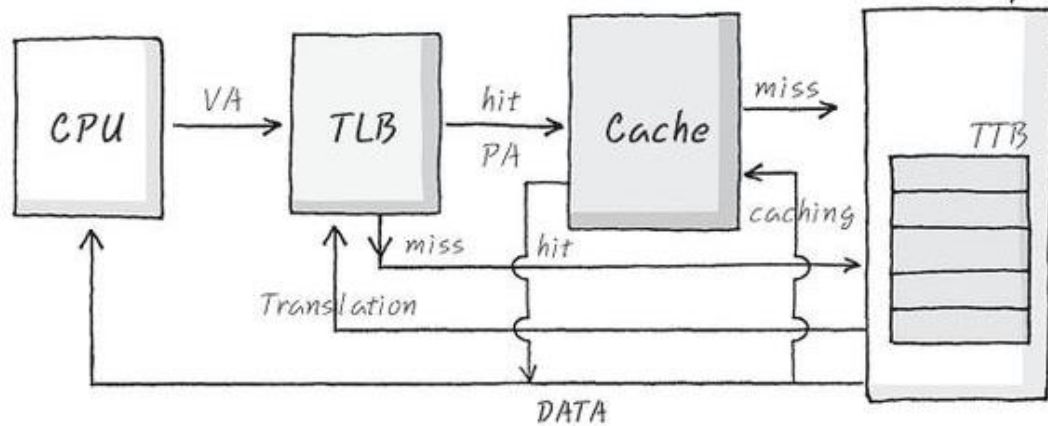
CPU Cache



〈그림 1〉 AMD/인텔의 최신 멀티 코어 프로세서 구조

Data Access

CPU Cache



Cache Size

CPU Cache



[인텔 코어 i7-9700K 커피레이크 리프레시](#)

최저 487,730원

종류 CPU

등록 2018.10.

리뷰 ★★★★★ 4.8 / 5 (전체 301건)

속성 인텔-코어 : 커피레이크-R | 코어 형태 : 옥타(8) 코어 | 동작 클럭 : 3.6GHz | 소켓 : 인텔-소켓1151v2 | 제조 공정 : 14nm |

상품구성

제품정보

쇼핑몰리뷰

DataLab.

[전체보기 →](#)

• 인텔-코어 ----- **커피레이크-R**

• 동작 클럭 ----- **3.6GHz**

• 터보클럭속도 ----- **4.90GHz**

• 터보기술 ----- **터보부스트**

• L1 캐시 ----- **256KB x 2**

• L3 캐시 ----- **12MB**

• 코어 형태 ----- **옥타(8) 코어**

• 소켓 ----- **인텔-소켓1151v2**

• 스레드 ----- **8스레드**

• 연산 체계 ----- **64bit**

• L2 캐시 ----- **2MB**

• 시스템버스 ----- **8**



BUS

신호 이동 통로

역할

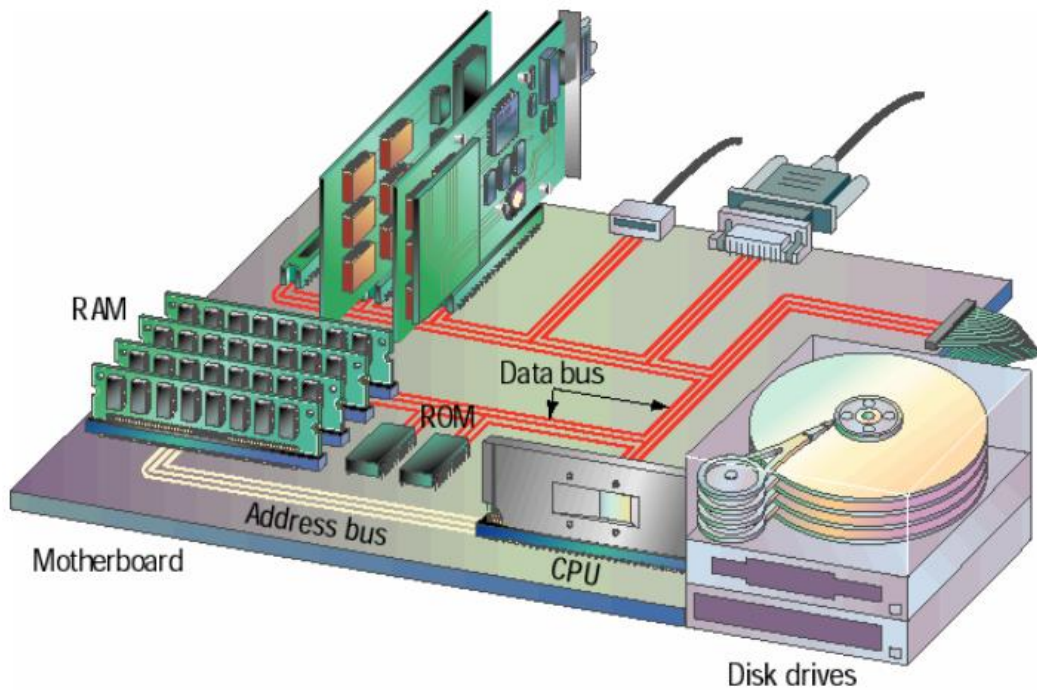
BUS



- 전기 신호가 오고 가는 통로
- 모든 신호는 BUS를 통해 전달
- 신호: 주소, 데이터, 제어 신호 등
- 컴퓨터 안의 Component간, 컴퓨터 간에 데이터를 전송하는 통신시스템
- 하드웨어 부품(선, 회로 등) + 통신 프로토콜 + 소프트웨어를 총칭

구조

BUS

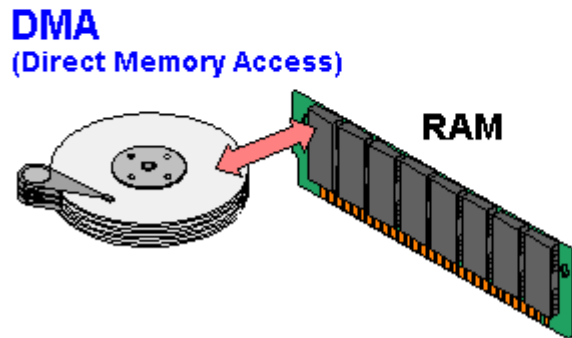
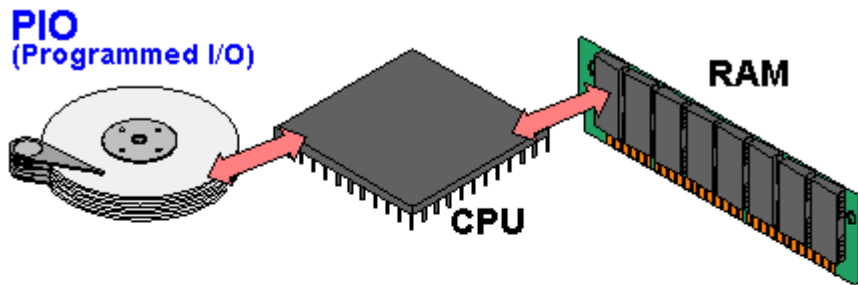


PIO & DMA

BUS



- PIO (Programmed I/O) => Polling과 Interrupt 사용
- DMA (Direct Memory Access) => Interrupt 사용

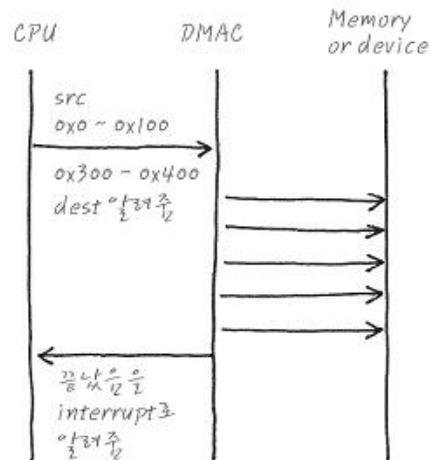
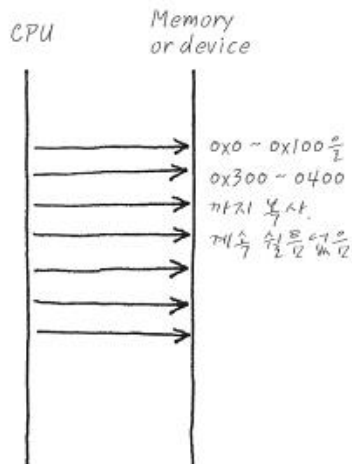
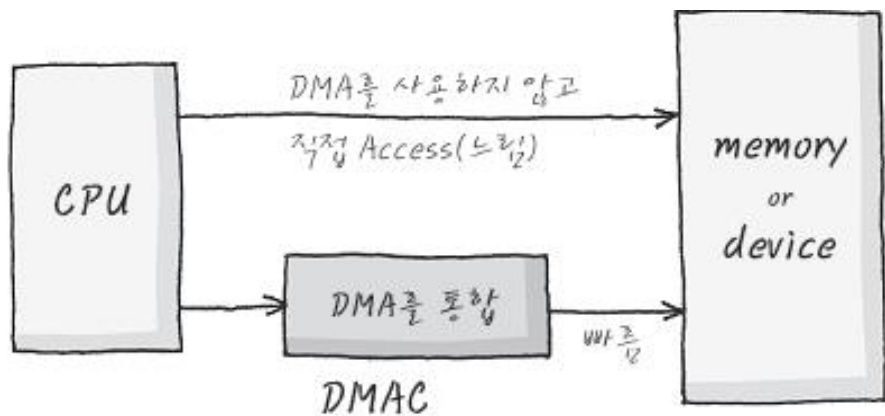


PIO & DMA

BUS



- PIO (Programmed I/O)
- DMA (Direct Memory Access)

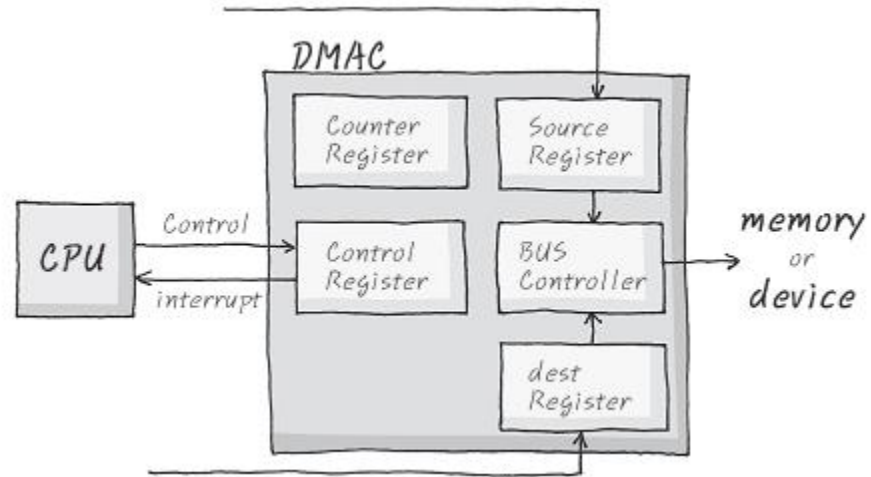


DMAC

BUS

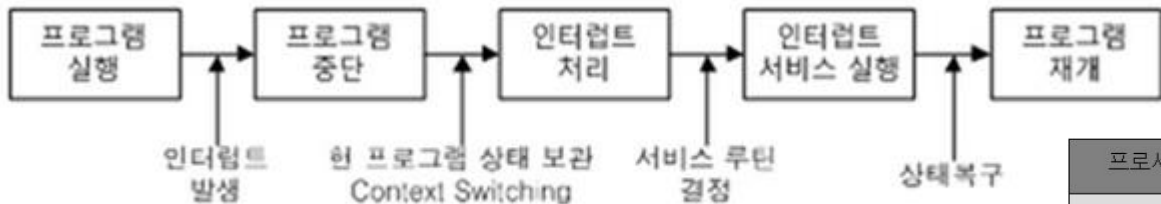


- DMAC (DMA Controller)

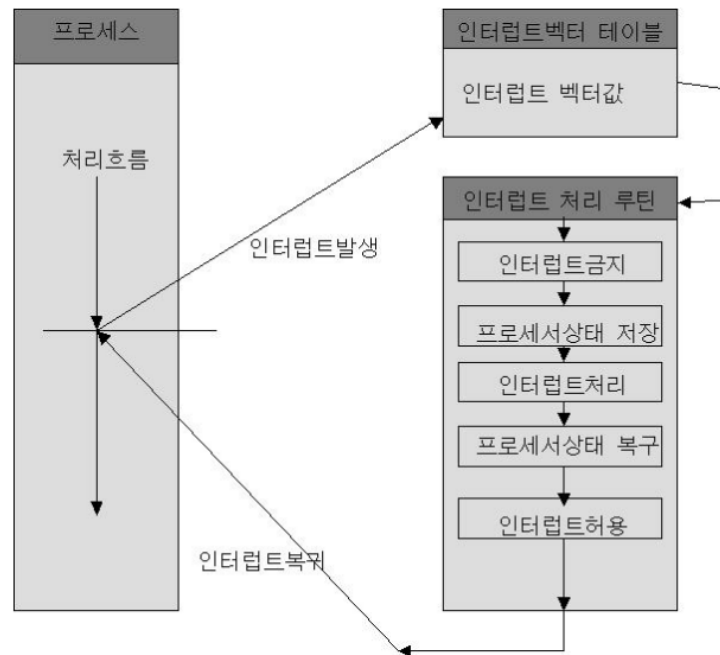


Interrupt

BUS



▶ 인터럽트 동작 원리



A close-up, slightly blurred photograph of a person's hands typing on a laptop keyboard. The person has dark red nail polish. A large, semi-transparent white circle is centered over the image, containing the text 'Thank you' and its Korean equivalent. To the right of the circle, there are three red circles of varying sizes, resembling a bubble or a decorative element.

Thank you

피드백이나 질문 부탁드립니다