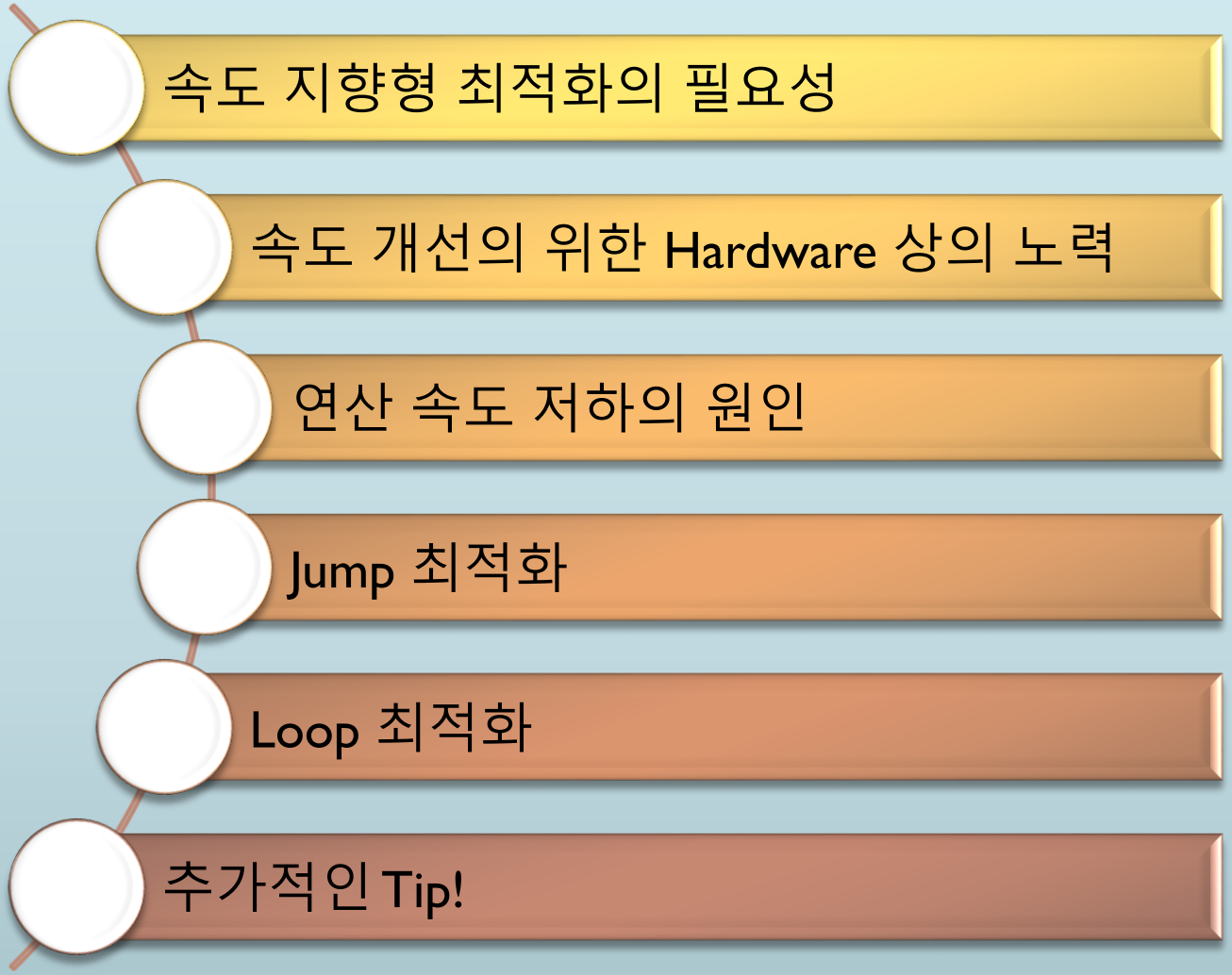
C Language환경에서의 속도 지향형 범용 코드 최적화 기법

유 용길

ISL Image System Laboratory

목차

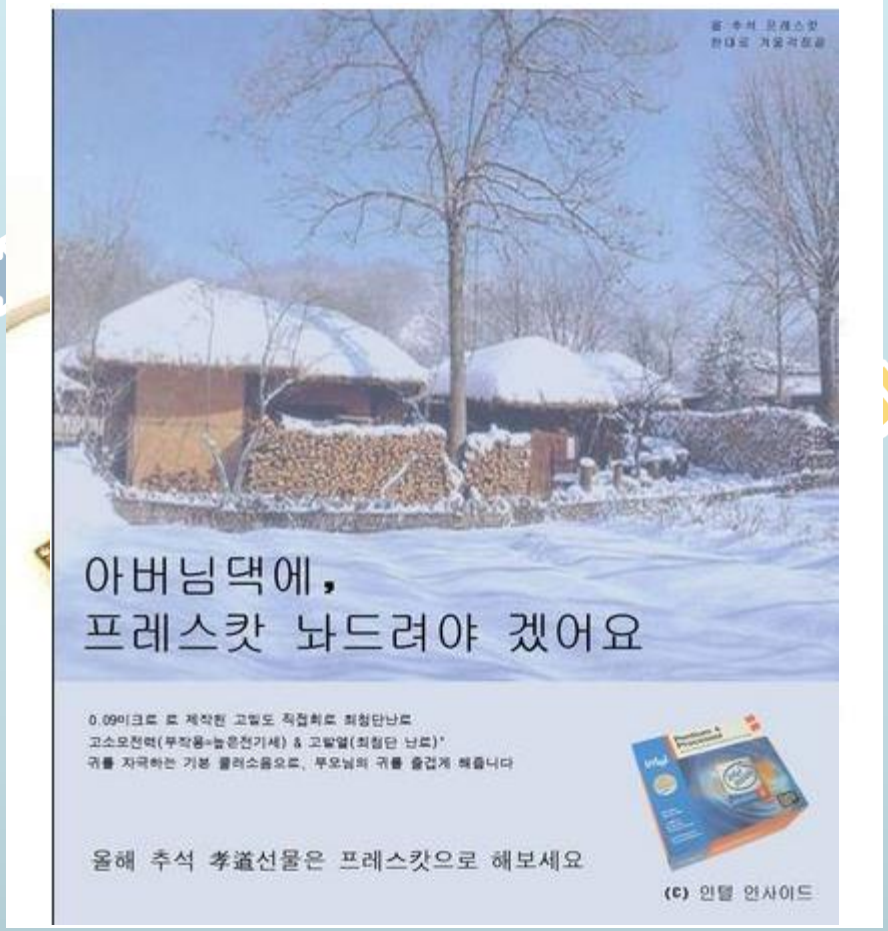

속도 지향형 최적화의 필요성
속도 개선의 위한 Hardware 상의 노력
연산 속도 저하의 원인
Jump 최적화
Loop 최적화
추가적인 Tip!

속도 지향형 최적화의 필요성

초기 컴퓨터의

프로그램 최적화

- 프로그램 크기 지향형 최적화.
  →메모리 가격이 매우 비쌈.
- 코드 길이를 줄여 Text 영역을 Save.



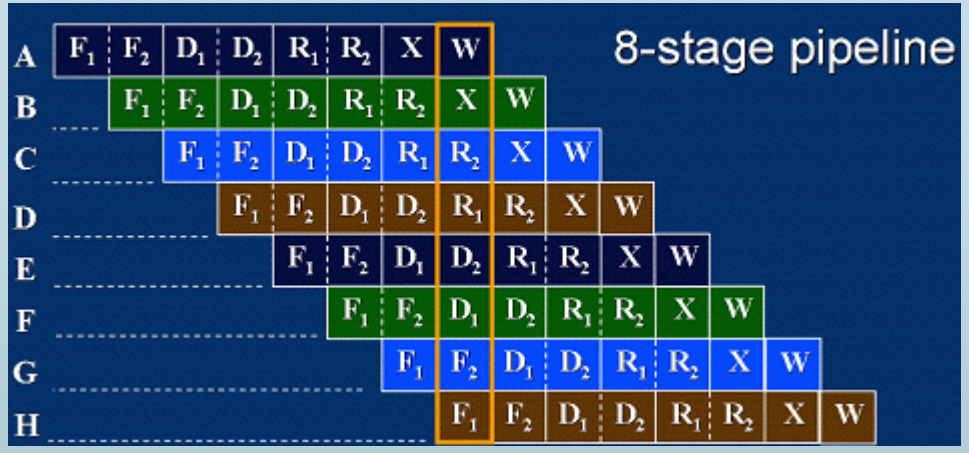
속도 지향형 최적화

- 대용량 메모리
  → 크기 지향형 최적화의 필요성이 약해짐.
- 코어 하나의 연산능력이 한계를 드러냄.
  →주어진 Hardware 조건에서 빠른 속도를 내야함.

속도개선의 위한 Hardware 상의 노력

- 명령어 Pipeline
  - 명령어 처리 구조

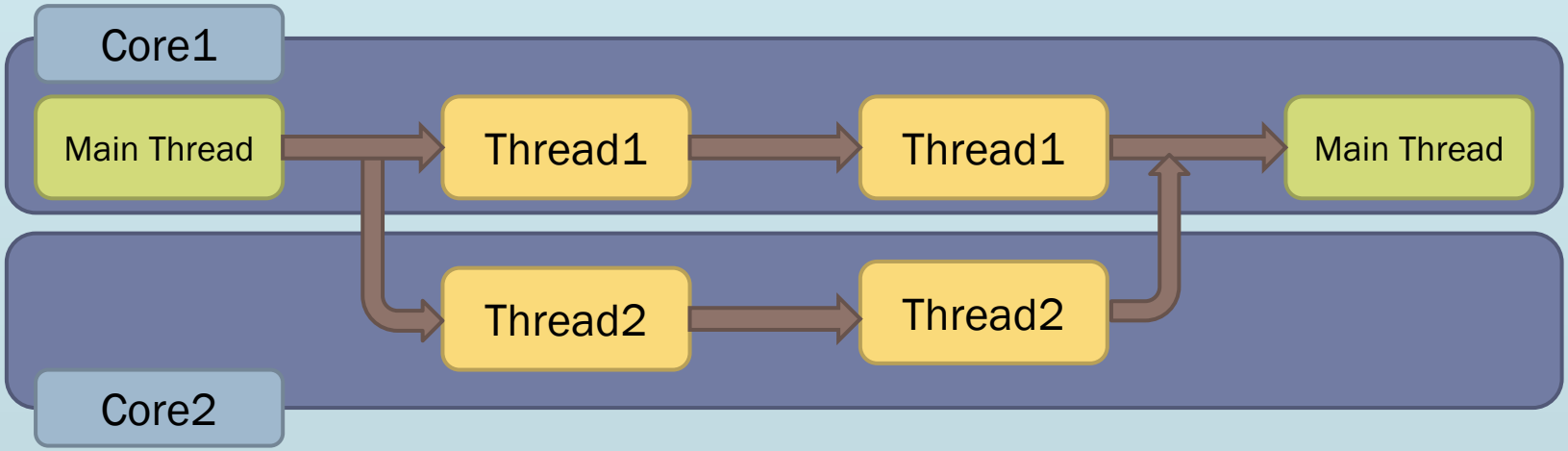
  - Pipeline Example



(Fetch : 2clock, Decode : 2clock, Read : 2clock, Excute : 1clock, Write 1clock)

속도개선의 위한 Hardware 상의 노력

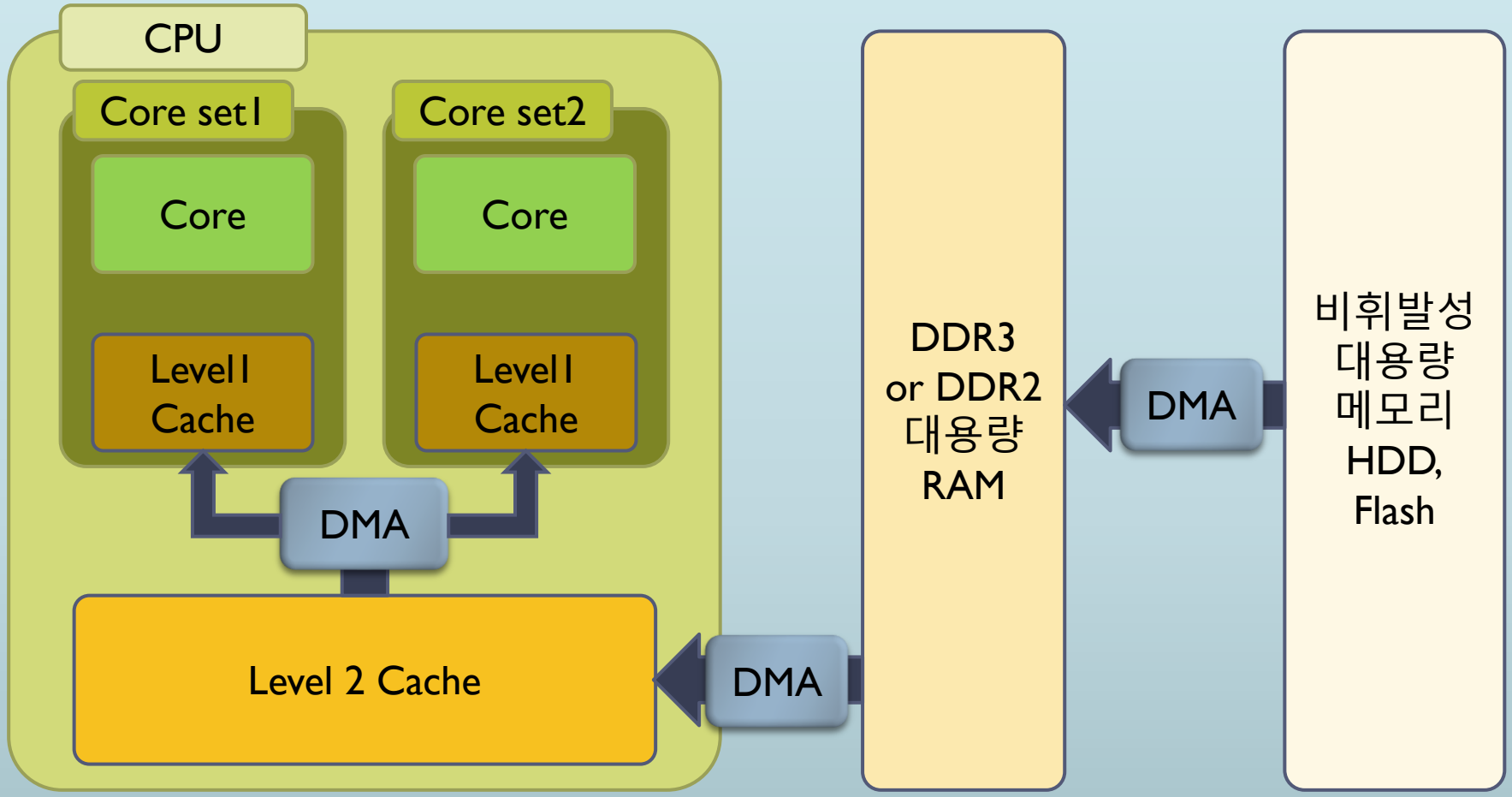
- Multi-Core
  - Multi-Core 속도 향상 원리



  - Multi-Core 의 효과
    - Thread로 분기 가능한 연산/전체 연산량 의 크기에 따라 달라진다.
    - 일반적으로 Dual-Core일 경우 1.4~1.6배 성능향상을 기대.

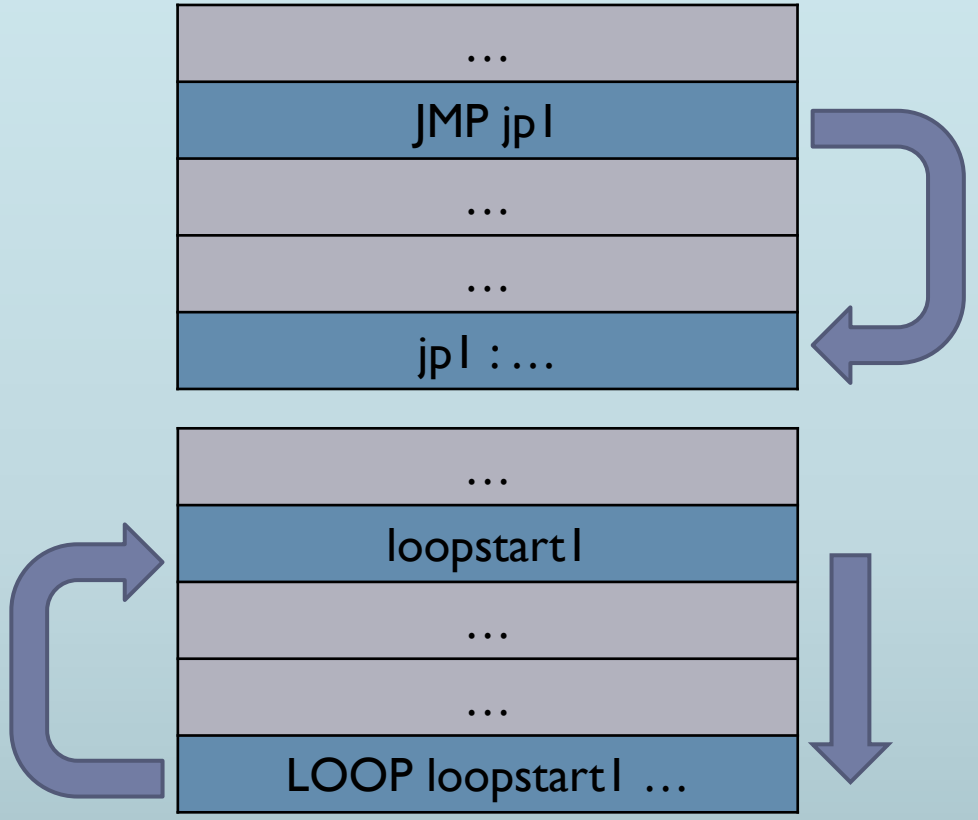
속도개선의 위한 Hardware 상의 노력

- Cache Memory & DMA (메모리가 밝은 색상 일수록 접근 속도가 느림)



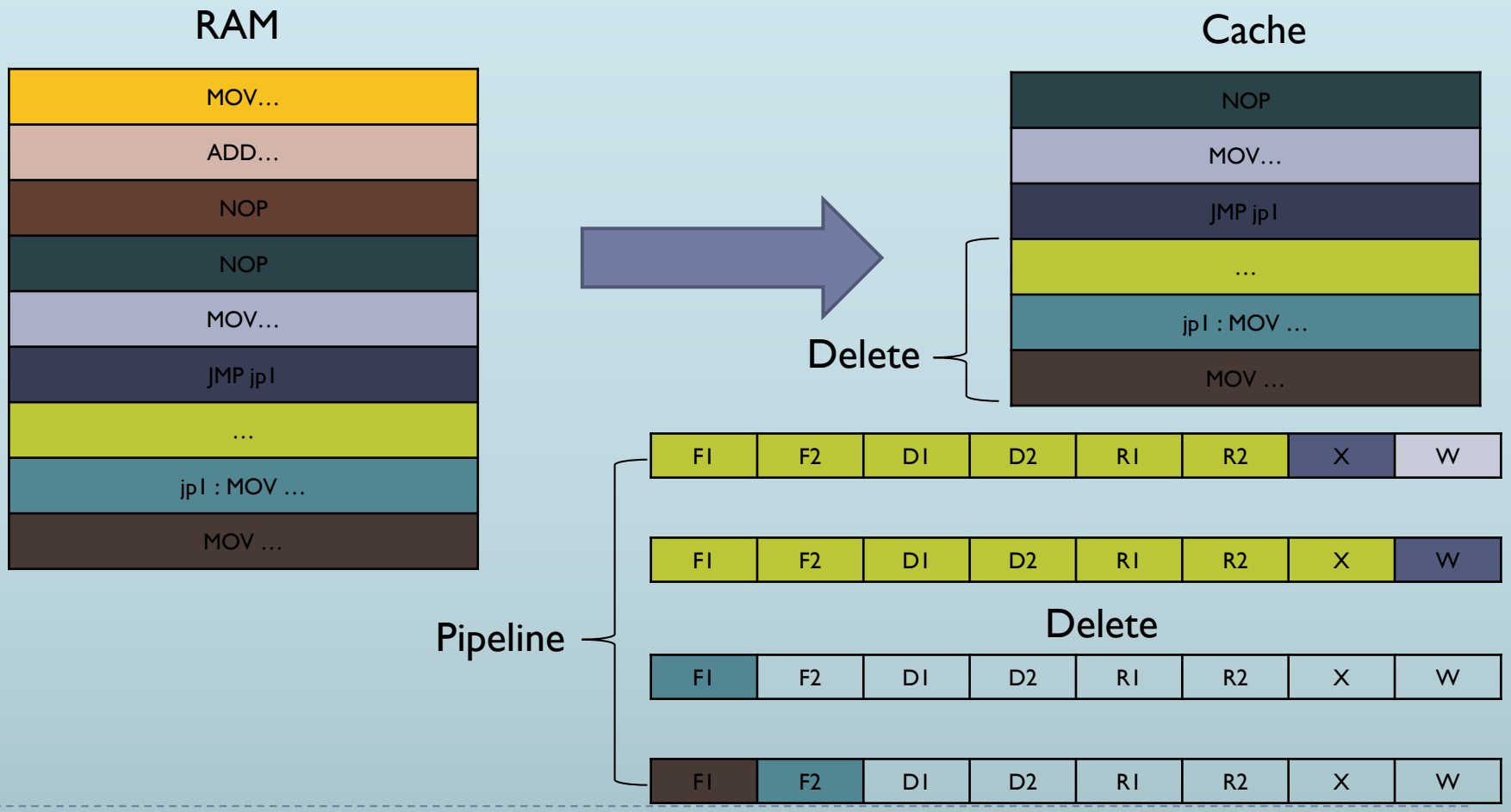
연산 속도 저하의 원인

- 기본 원리
  - 메모리 접근 속도
  - Cache와 명령어 Pipeline 사용률
- Jump & Call
  - 메모리의 불특정 위치로 PC를 지정
  - Jump - 분기 구문에서 발생
    - If, else
    - switch, case
    - goto:
  - Call - 함수 호출에서 발생
- Loop
  - 조건에 따라 반복 분기
  - 반복문에서 발생
    - for
    - while



연산 속도 저하의 원인

- 분기와 Cache 및 Pipeline 사용률과의 관계



Jump 최적화

- ~~switch case~~
- ~~goto~~
- if, else
  - 연속적인 if 구문 사용 지양

```
if(flag == 0) …;
if(flag == 1) …;
if(flag == 2) …;
```

```
if(flag == 0) …;
else If(flag == 1) …;
else …;
```

  - 실행 빈도가 높은 순서대로 배치

(flag == 0 → 10%, flag == 1 → 50%, flag == 2 → 40%)

```
if(flag == 1) …;
else if(flag == 2) …;
else …;
```

Call 최적화

- 함수의 단일화
  - 자주 호출되는 함수를 코드에 직접 삽입

```
Int add(int a, intb)
{
   int c;
   c = a+ b;
   return c;
}

void main(void)
{
   int a, b, c;
   …
   a = 1;
   b = 2;
   c = add(a, b);
   …
}
```

```
void main(void)
{
   int a, b, c;
   …
   a = 1;
   b = 2;
   c = c = a+ b;
   …
}
```

Call 최적화

- inline 함수 활용

```
Int add(int a, intb)
{
   int c;
   c = a+ b;
   return c;
}

void main(void)
{
   int a, b, c;
   …
   a = 1;
   b = 2;
   c = add(a, b);
   …
}
```

```
inline Int add(int a, intb)
{
   int c;
   c = a+ b;
   return c;
}

void main(void)
{
   int a, b, c;
   …
   a = 1;
   b = 2;
   c = add(a, b);
   …
}
```

Loop 최적화

- Unrolled loop 사용
  - Loop 반복 횟수의 상수화

```
count = 10;
for(int i = 0; i <count; i++)
{
                    …
}
```

```
#define CNT 10
…
for(int i = 0; i < CNT; i++)
{
                    …
}
```

  - Loop를 풀어 헤치자!

```
#define CNT 2
…
for(int i = 0; i < CNT; i++)
{
                    a1[i] = a2[i] + a3[i];
}
```

```
a1[0] = a2[0] + a3[0];
a1[1] = a2[1] + a3[1];
a1[2] = a2[2] + a3[2];
```

추가적인 Tip!

- 상수 나눗셈 지양
  - 나눗셈은 전용 연산기가 없음.
  - 수치해석적 기법으로 산출.

```
Float a, b;
…
a=b/50;
…
```

```
Float a, b;
…
a=b*0.02;
…
```

- 산술 연산 or 논리 연산 < 비트 연산
  - 비트 연산은 단일 명령어로 처리될 확률이 높음.

```
float a, b, c;
…
a=b*2;
If(c == 0) c = 1;
else c = 0;
…
```

```
float a, b, c;
…
a=b<<1;
c^=1;
…
```

추가적인 Tip!

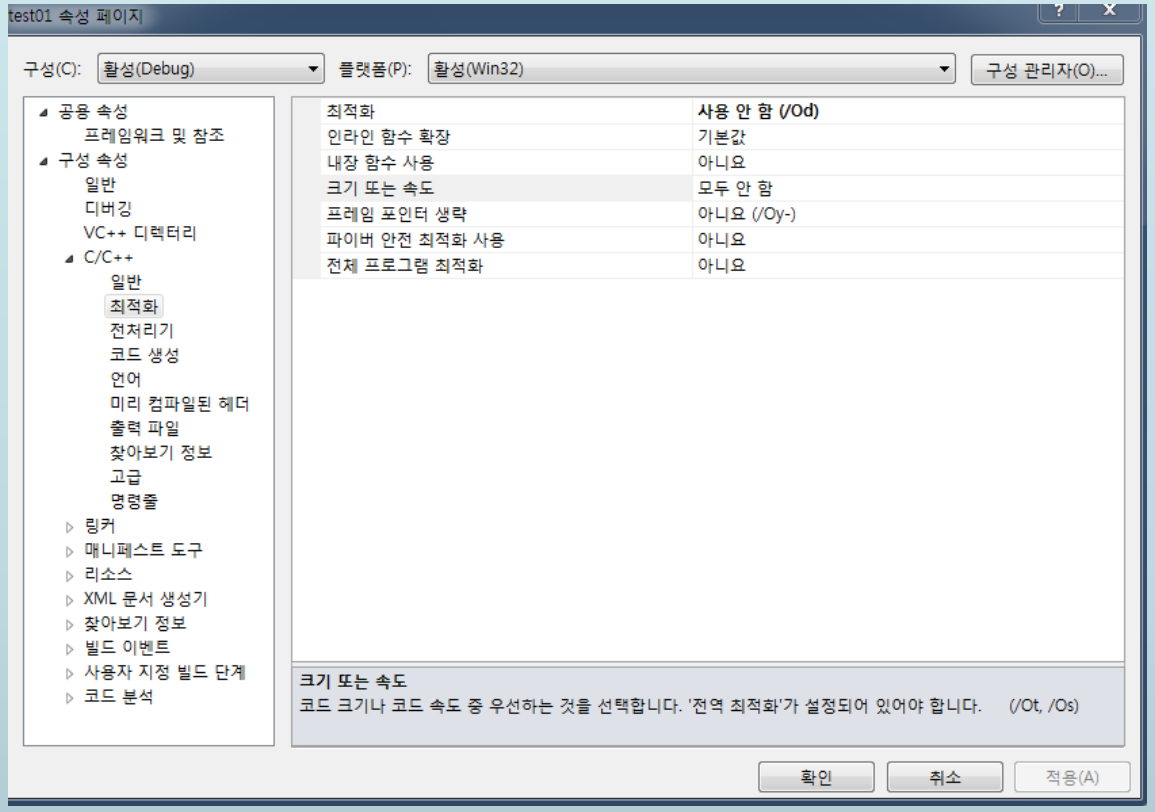
- 메모리 이동 작업 최소화
  - 사용자 프로그램에 의한 메모리 이동은 DMA에게 맡겨질 확률이 낮음.
  - 메모리 이동 없이 연산하게 코딩.

```
…
for(int i = 7; I > 0; i--)
{
   a[i] = a[i-1];
}
a[0] = new_value;
for(int i = 0; I < 8; i++)
{
   b[i] = a[i] * coef[i];
}
…
```

```
…
a[count & 0x03] = new_value;
for(int i = 0; I < 8; i++)
{
   b[i] = a[(count – i) & 0x03] * coef[i];
}
…
```

추가적인 Tip!

- Compiler의 최적화 옵션 사용
  - Compiler 고유의 최적화 옵션 중 Speed 옵션을 선택



- 최적화는 항상 개발의 마지막 단계에서 활용!

Q&A