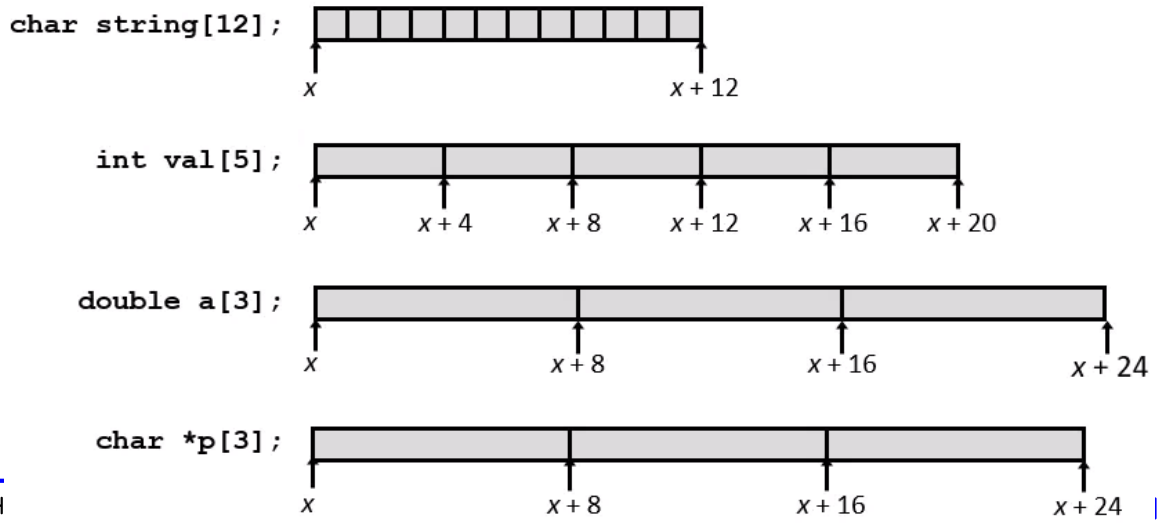
3장. 프로그램의 기계수준 표현

3.8 배열의 할당과 접근

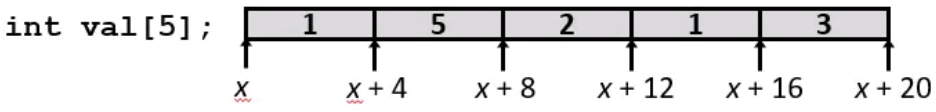
배열 할당 (Array Allocation)

- T A[L];
  - 자료형 T와 길이 L의 배열
  - L*sizeof(T) 바이트가 메모리에 연속적으로 할당



배열 접근 (Array Access)

- T A[L];
  - 자료형 T와 길이 L의 배열
  - 식별자 A는 배열의 시작 원소 0의 포인터: TYPE T*

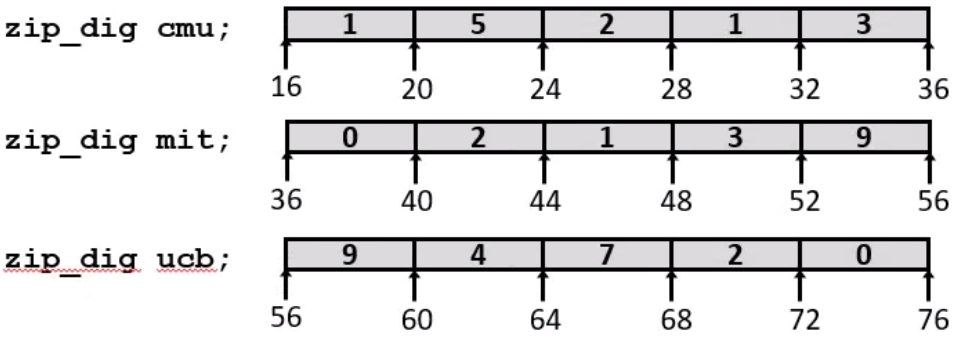


Reference	Type	Value
val[4]	int	3
val	int *	x
val+1	int *	x + 4
&val[2]	int *	x + 8
val[5]	int	??
*(val+1)	int	5
val + i	int *	x + 4 i

배열 예

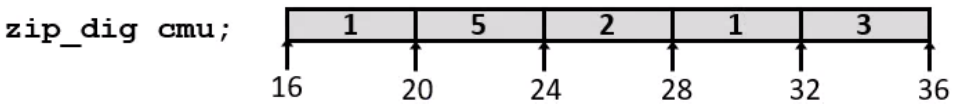
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- zip_dig cmu 선언은 int cmu[5]와 같음
- 위 예에서는 20바이트 블럭이 메모리에 연속적으로 할당되었다고 가정

배열 접근 예




```
int get_digit
  (zip_dig z, int digit)
{
  return z[digit];
}
```

- %rdi는 배열의 시작 주소
- %rsi는 배열의 인덱스
- 배열 값의 주소: %rdi + 4*%rsi
- 주소지정 모드: (%rdi, %rsi, 4)

IA32

```
  # %rdi = z
  # %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

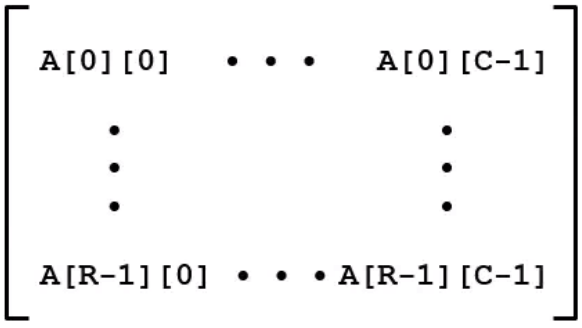
배열 루프 예

```
void zincr(zip_dig z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

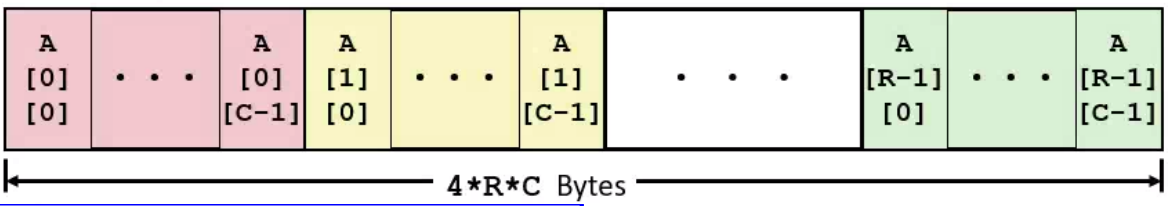
```
  # %rdi = z
  movl    $0, %eax          #   i = 0
  jmp     .L3               #   goto middle
.L4:                        # loop:
  addl    $1, (%rdi,%rax,4) #   z[i]++
  addq    $1, %rax          #   i++
.L3:                        # middle
  cmpq    $4, %rax          #   i:4
  jbe     .L4               #   if <=, goto loop
  rep; ret
```


다중 배열 (다차원 배열) (Multidimensional Array, Nested Array)

- T A[R][C];
  - 자료형 T의 2차원 배열
  - R 행, C 열
  - 자료형 T의 요소는 K 바이트
- 배열의 크기
  - R * C * K 바이트
- 배치
  - 행우선 순서(Row-Major Order)



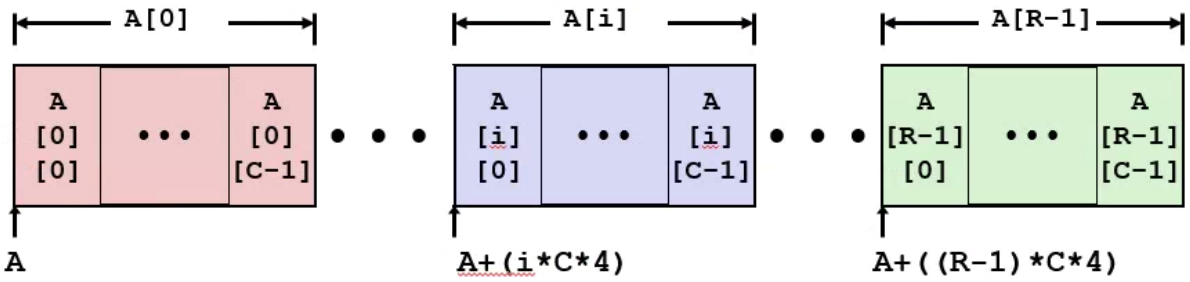

```
int A[R][C];
```



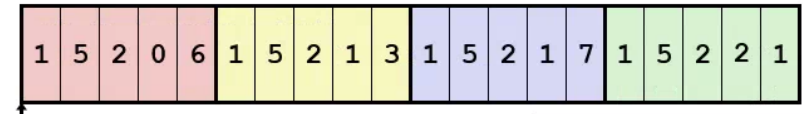
다중 배열의 행 접근

- 행 벡터 (Row Vectors)
  - A[i] 는 C개의 요소들을 갖는 배열
  - 각 요소는 자료형 T의 K 바이트
  - 배열의 시작주소는 A + i*(C*K)

```
int A[R][C];
```



다중 배열의 행 접근 코드




```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

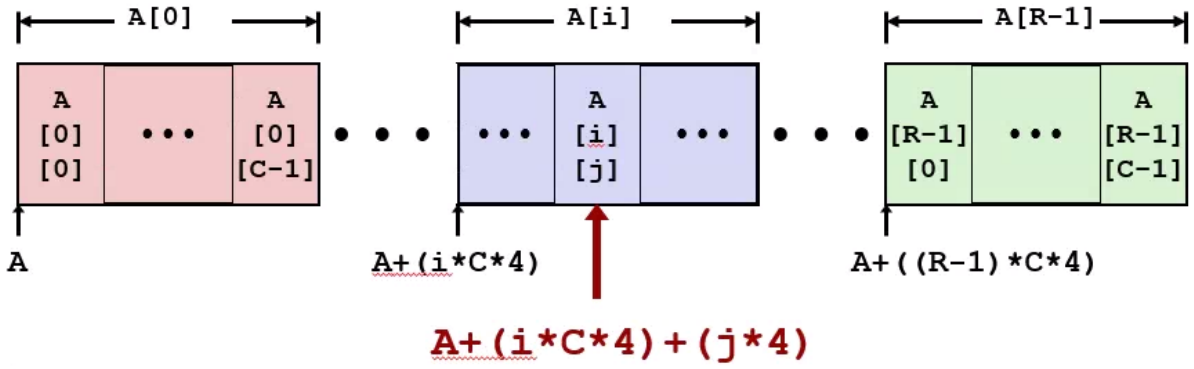
```
# %rdi = index
 leaq (%rdi,%rdi,4),%rax # 5 * index
 leaq pgh(,%rax,4),%rax  # pgh + (20 * index)
```

- 행 벡터
  - pgh[index]는 5개의 정수형 배열
  - 시작주소는 pgh+(index*5*4)
- 기계어 코드
  - 행벡터 시작주소를 계산하고 리턴
  - pgh+ 4*(index+4*index)로 계산

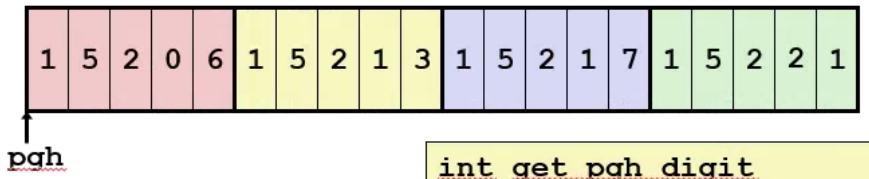
다중 배열의 요소 접근

- 배열 요소 (Array Elements)
  - A[i][j]는 K 바이트 크기의 자료형 T의 요소
  - 주소는 A+i*(C*K)+j*K = A+(i*C+j)*K

```
int A[R][C];
```



다중 배열의 요소 접근 코드




```
int get_pgh_digit
   (int index, int dig)
{
  return pgh[index][dig];
}
```

```
leaq   (%rdi,%rdi,4), %rax    # 5*index
addl   %rax, %rsi             # 5*index+dig
movl   pgh(,%rsi,4), %eax     # M[pgh + 4*(5*index+dig)]
```

- 배열 요소
  - pgh[index][dig]는 정수형(int)
  - 주소는 pgh + 20*index + 4*dig
    - pgh + 4 * (5*index + dig)

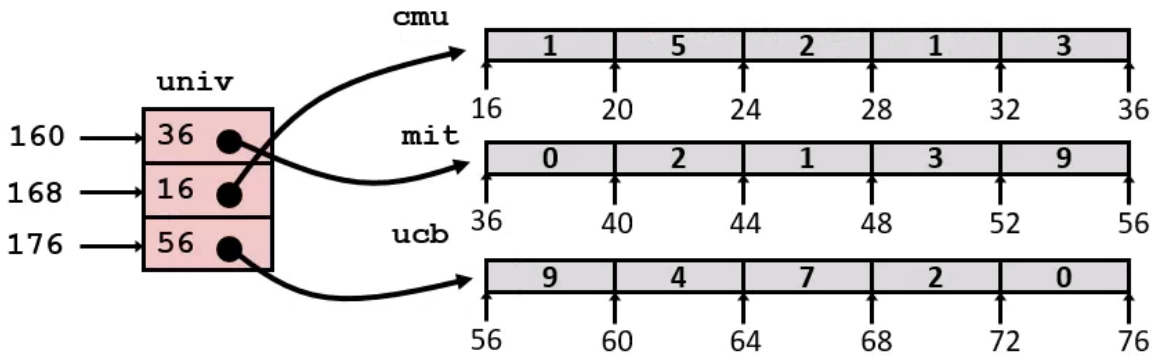
다중-수준 배열 (Multi-Level Array) 예

```
#define ZLEN 5
typedef int zip_dig[ZLEN];
```

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

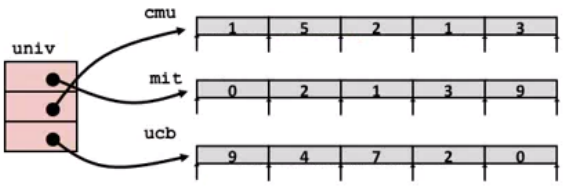
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- 변수 univ
  - 3개의 요소를 갖는 배열
  - 각 요소는 포인터
    - 8 바이트
  - 각 포인터는 정수형의 배열을 가리킴



다중-수준 배열의 요소 접근

```
int get_univ_digit
   (size_t index, size_t digit)
{
   return univ[index][digit];
}
```




```
   salq    $2, %rsi              # 4*digit
   addq    univ(,%rdi,8), %rsi   # p = univ[index] + 4*digit
   movl    (%rsi), %eax          # return *p
   ret
```

- 계산
  - 요소 접근 Mem[Mem[univ+8*index]+4*digit]
  - 메모리를 두 번 읽음
    - 첫 번째는 행 배열의 포인터
    - 다음은 배열 내의 요소 접근

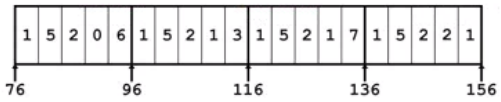
배열 요소 접근

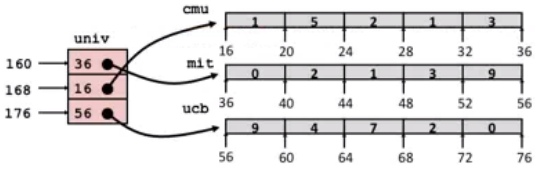
Nested array

```
int get_pgh_digit
   (size_t index, size_t digit)
{
  return pgh[index][digit];
}
```

Multi-level array

```
int get_univ_digit
   (size_t index, size_t digit)
{
  return univ[index][digit];
}
```





- C 언어는 비슷해 보이지만 주소 계산 방법은 차이가 있음

```
Mem[pgh+20*index+4*digit]    Mem[Mem[univ+8*index]+4*digit]
```

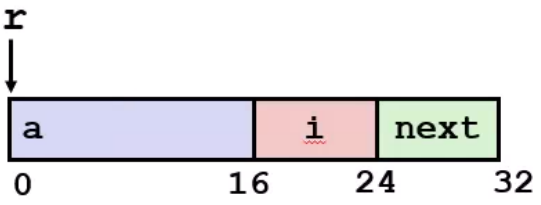

과제 3-9: 다중 배열의 gdbgui 실행 (실습과제)

- 가상머신에서 앞의 다중 배열, 다중-수준 배열을 아래와 같이 각각 C 프로그램을 작성
  - 다중 배열 버전
    - int pgh[][5] = { { 1, 5, 2, 1, 3 }, { 0, 2, 1, 3, 9 }, { 9, 4, 7, 2, 0 } };
    - get_pgh_digit() 함수 사용
  - 다중-수준 배열
    - int cmu[] = { 1, 5, 2, 1, 3 };
      int mit[] = { 0, 2, 1, 3, 9 };
      int ucb[] = { 9, 4, 7, 2, 0 };
      int *univ[] = { mit, cmu, ucb};
    - get_univ_digit() 함수 사용
- 각각 컴파일 한 후 실행파일을 gdbgui로 실행하여 주요 단계 실행 및 상태를 추적 캡쳐하고 설명
  - 배열의 요소 값이 저장된 메모리 주소와 내용 확인하고 내용도 캡쳐
- 각 배열 요소 접근 어셈블리 코드를 분석하고 비교
  - 앞에서 소 개된 코드와의 차이도 설명

3.9 이기종 자료구조

구조체 (Structure) 표현

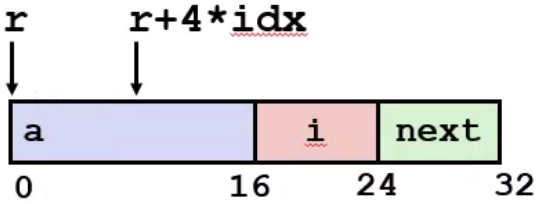
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- 구조체는 서로 다른 유형(자료형)의 개체(필드)들을 메모리 블럭에 저장
  - 모든 필드들이 연속된 메모리 공간에 저장
  - 컴파일러는 각 필드의 바이트 오프셋을 가리키는 각 구조체 유형에 관한 정보를 관리

구조체 멤버의 포인터

```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- 배열 요소의 포인터 생성 예
  - 컴파일 시 구조체의 각 멤버 오프셋이 결정됨
  - r + 4*idx 로 계산

```
int *get_ap
 (struct rec *r, size_t idx)
{
  return &r->a[idx];
}
```

```
  # r in %rdi, idx in %rsi
  leaq   (%rdi,%rsi,4), %rax
  ret
```

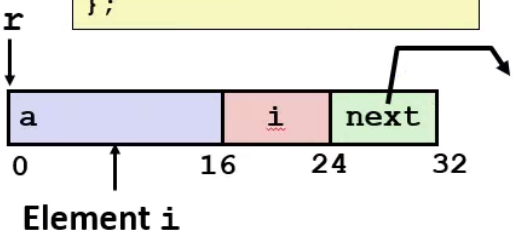

연결 리스트 예

Following Linked List

- C Code

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

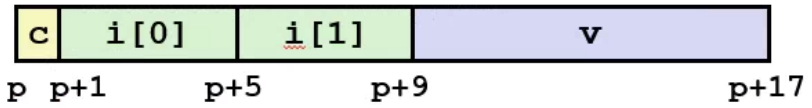


Register	Value
%rdi	r
%rsi	val

```
.L11:                           # loop:
  movslq  16(%rdi), %rax        #   i = M[r+16]
  movl    %esi, (%rdi,%rax,4)   #   M[r+4*i] = val
  movq    24(%rdi), %rdi        #   r = M[r+24]
  testq   %rdi, %rdi            #   Test r
  jne     .L11                  #   if !=0 goto loop
```

데이터의 정렬 (Alignment)

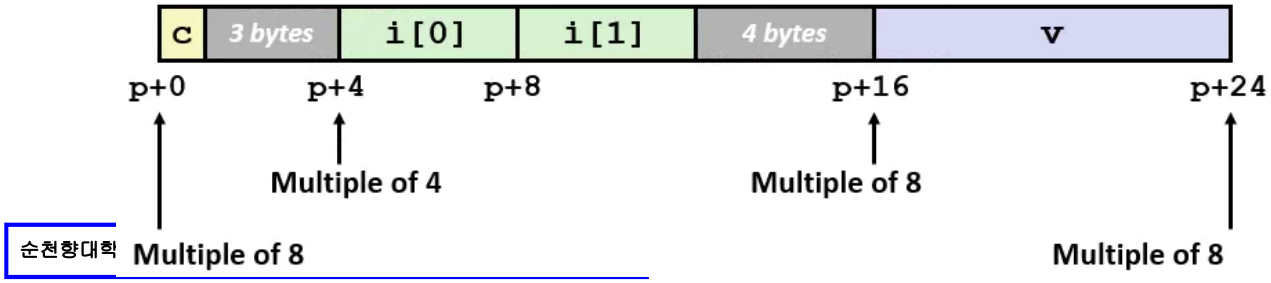
❑ 정렬되지 않은 데이터 (Unaligned Data)




```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

❑ 정렬된 데이터 (Aligned data)

- 기본 자료형이 K 바이트
- 주소는 K 바이트의 배수



정렬 규칙

❑ 정렬된 데이터 (Aligned data)

- 기본 자료형이 K 바이트
- 주소는 K 바이트의 배수
- 일부 머신은 정렬이 필수
  - x86-64는 권장 사항

❑ 정렬이 필요한 이유

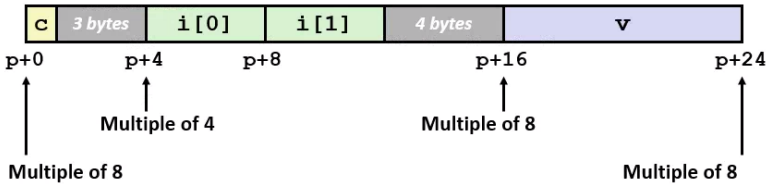
- 일반적으로 시스템은 4 또는 8 바이트 단위로 메모리 접근

❑ 컴파일러

- 올바른 정렬을 위해 빈 공간을 삽입

자료형의 정렬 규칙 (x86-64)

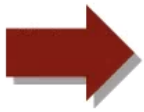
- 1 바이트: char
  - K=1, 메모리 주소에 어떤 제한도 없음
- 2 바이트: short
  - K=2, 메모리 주소는 2의 배수이어야 함
- 4 바이트: int, float
  - K=4, 메모리 주소는 4의 배수이어야 함
- 8 바이트: double, long, char *, ...
  - K=8, 메모리 주소는 8의 배수이어야 함
- 16 바이트: long double (리눅스의 GCC)
  - K=16, 메모리 주소는 16의 배수이어야 함



메모리 공간 절약

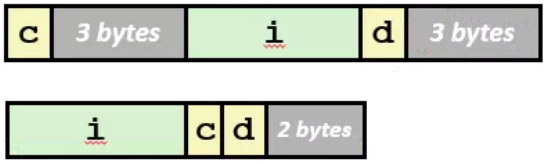
- 구조체에서 자료형이 큰 데이터 먼저 기술

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```



```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

- 효과 (K=4)



3.11 부동소수점 코드

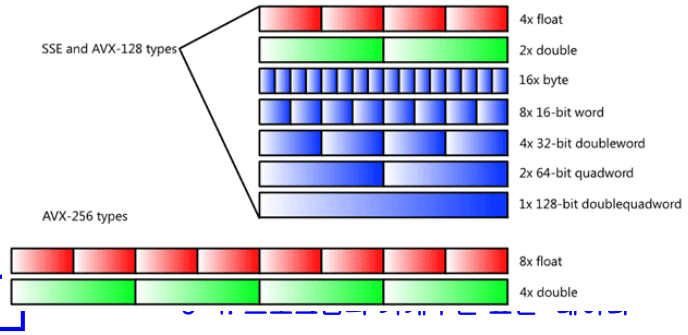
부동소수점 아키텍처

- 부동소수점 아키텍처는 부동소수점 데이터 연산을 지원
  - 부동 소수점 레지스터
  - 부동소수점 명령어
- x86-64 부동소수점 아키텍처
  - 그래픽과 영상처리를 위한 미디어 명령어 지원
  - 미디어 명령어들은 SIMD (Single Instruction Multiple Data) 명령어
    - 다 수의 서로 다른 데이터 (벡터 데이터) 들이 동일 연산을 병렬로 수행
    - 멀티미디어 응용 등과 같은 높은 데이터 수준 병렬성(data-level parallelism)을 갖는 응용에 효율적

x86-64 미디어 명령어

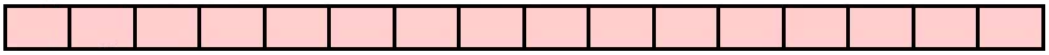
x86-64 미디어 명령어 발전 단계

- MMX (MultiMedia eXtension)
  - 1997년 펜티엄 프로세서
  - 64 비트의 MM 레지스터
- SSE (Streaming SIMD Extension)
  - 1999년 펜티엄3 프로세서
  - 128비트의 XMM 레지스터
- AVX (Advanced Vector Extension)
  - 2010년 Core i 시리즈
  - 현재는 AVX2
  - 256비트의 YMM 레지스터
    - 8개의 32비트 값
    - 4개의 64비트 값
  - <그림 3.45> 미디어 레지스터

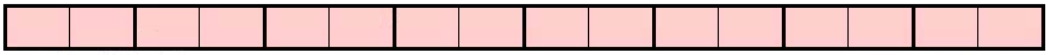


XMM 레지스터

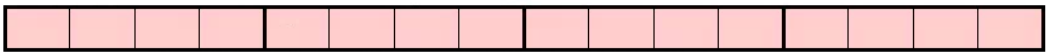
- 16 total, each 16 bytes
- 16 single-byte integers



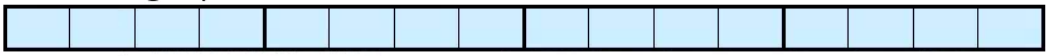
- 8 16-bit integers



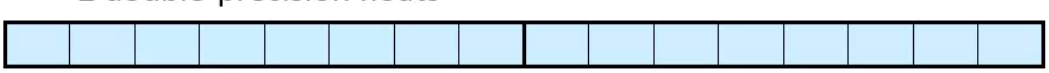
- 4 32-bit integers



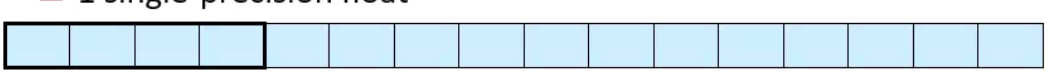
- 4 single-precision floats



- 2 double-precision floats

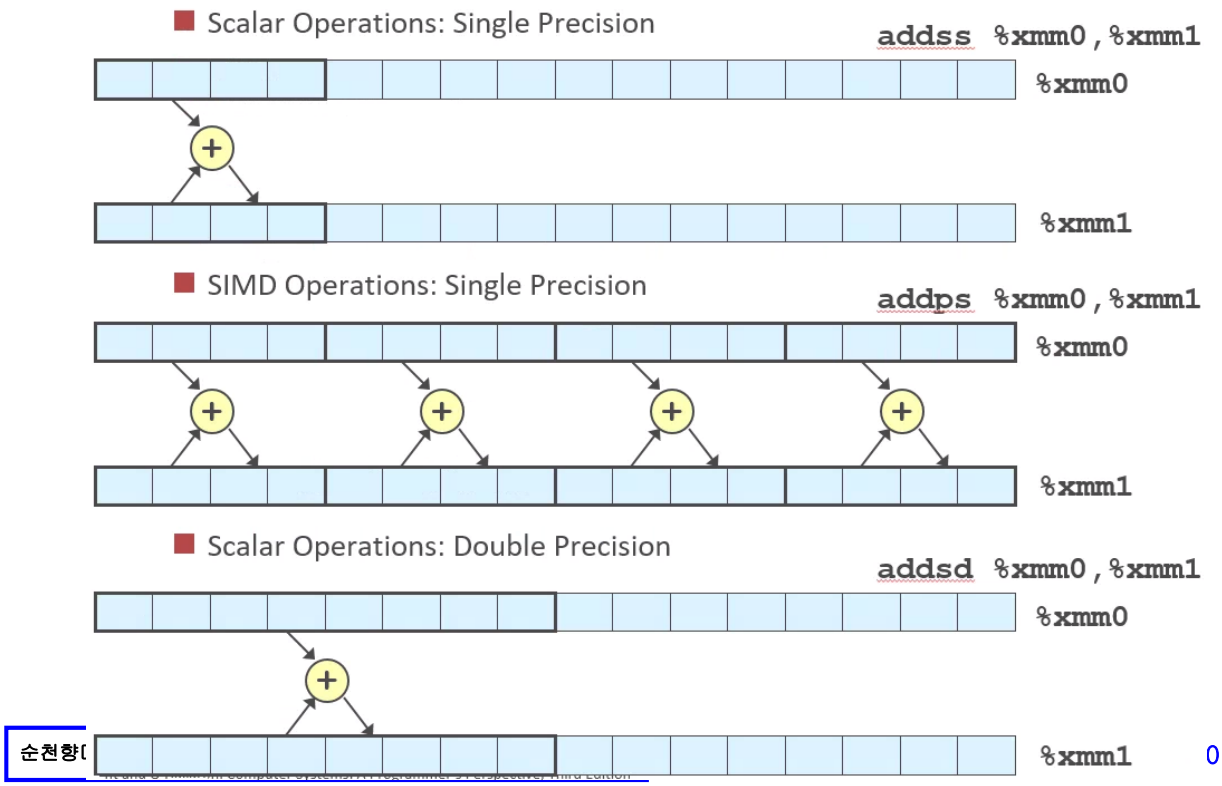


- 1 single-precision float



- 1 double-precision float

스칼라 및 SIMD 연산



부동소수점 코드 – 프로시저

- 인수들은 %xmm0, %xmm1, ... 레지스터로 전달
- 리턴 값은 %xmm0 레지스터
- 모든 XMM 레지스터는 호출자 저장 (caller-saved)

```
float fadd(float x, float y)
{
    return x + y;
}
```

```
double dadd(double x, double y)
{
    return x + y;
}
```

```
  # x in %xmm0, y in %xmm1
  addss   %xmm1, %xmm0
  ret
```

```
 # x in %xmm0, y in %xmm1
  addsd   %xmm1, %xmm0
  ret
```

부동소수점 코드 – 메모리 참조

- 정수나 포인터 인수들은 정수 레지스터로 전달
- 부동소수점 데이터는 XMM 레지스터로 전달
- XMM 레지스터 간 이동이나 XMM과 메모리 간의 이동 시에는 서로 다른 명령어 사용
  - movapd (move aligned, packed double precision)
  - movsd (move scalar double precision)

```
double dincr(double *p, double v)
{
    double x = *p;
    *p = x + v;
    return x;
}
```

```
  # p in %rdi, v in %xmm0
  movapd   %xmm0, %xmm1    # Copy v
  movsd    (%rdi), %xmm0   # x = *p
  addsd    %xmm0, %xmm1    # t = x + v
  movsd    %xmm1, (%rdi)   # *p = t
  ret
```


과제 3-10: dincr() 함수의 gdbgui 실행 (실습과제)

- 가상머신에서 앞의 dincr() 함수를 아래와 같이 C 프로그램을 작성하고 컴파일 한 후 실행파일을 gdbgui로 실행하여 메모리 및 레지스터 등의 상태 추적
  - 실행파일 생성을 위해 main() 함수 추가
  - 아래와 같이 컴파일하고 주요 단계 실행 및 상태를 추적 캡쳐하고 설명

요 약

요 약

- 배열 (Array)
  - 배열의 요소는 연속된 메모리 공간에 배치
  - 각 요소의 위치 지정을 위해 인덱스를 사용해 계산
- 구조체 (Structure)
  - 구조체의 각 요소들은 인접된 메모리 영역에 배치
  - 컴파일러가 계산한 오프셋을 사용하여 접근
  - 정렬을 위해 빈공간을 덧붙임
- 부동소수점
  - 데이터는 XMM 레지스터에 저장되고 연산

과 제

과제 3-9: 다중 배열의 gdbgui 실행 (실습과제)

- 가상머신에서 앞의 다중 배열, 다중-수준 배열을 아래와 같이 각각 C 프로그램을 작성
  - 다중 배열 버전
    - int pgh[][5] = { { 1, 5, 2, 1, 3 }, { 0, 2, 1, 3, 9 }, { 9, 4, 7, 2, 0 } };
    - get_pgh_digit() 함수 사용
  - 다중-수준 배열
    - int cmu[] = { 1, 5, 2, 1, 3 };
      int mit[] = { 0, 2, 1, 3, 9 };
      int ucb[] = { 9, 4, 7, 2, 0 };
      int *univ[] = { mit, cmu, ucb};
    - get_univ_digit() 함수 사용
- 각각 컴파일 한 후 실행파일을 gdbgui로 실행하여 주요 단계 실행 및 상태를 추적 캡쳐하고 설명
  - 배열의 요소 값이 저장된 메모리 주소와 내용 확인하고 내용도 캡쳐
- 각 배열 요소 접근 어셈블리 코드를 분석하고 비교
  - 앞에서 소 개된 코드와의 차이도 설명

과제 3-10: dincr() 함수의 gdbgui 실행 (실습과제)

- 가상머신에서 앞의 dincr() 함수를 아래와 같이 C 프로그램을 작성하고 컴파일 한 후 실행파일을 gdbgui로 실행하여 메모리 및 레지스터 등의 상태 추적
  - 실행파일 생성을 위해 main() 함수 추가
  - 아래와 같이 컴파일하고 주요 단계 실행 및 상태를 추적 캡쳐하고 설명