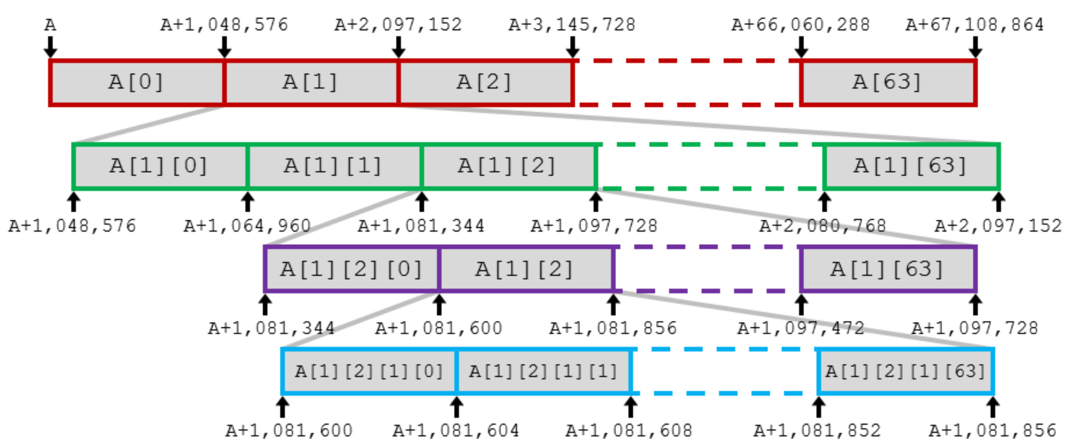
The HW/SW Interface

RISC-V
Composite Data Structures



Module Outline

- Data Alignment
- Arrays
- Structs
- Unions
- Module Summary

Type	64-bit RISC-V	
	Size	Alignment
bool	1	1
char	1	1
short	2	2
int	4	4
long	**8**	**8**
long long	8	8
__int128	16	16
void *	**8**	**8**

Data Alignment

Data Alignment

- Alignment of data
  - rules regarding the location of data in the memory of a computer system
    - physically required on some machines (ARM); advised on others (RISC-V, Intel)
  - different rules by processor and operating system
    - defined by the ABI (application binary interface) of a processor/OS
    - RISC-V ELF psABI Specification
  - (general) basic rule
    - primitive data type requires *K* bytes → memory address must be K-byte aligned
    - K-byte aligned = address divisible by K
  - example:
    - 4-byte integers must be located at memory addresses divisible by 4
      - valid addresses: 0, 4, 8, 12, 16, …
      - invalid addresses: 1,2,3,5,6,7,9,10,11,…

Data Alignment

- Motivation for aligning data
  - memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    Unit (not a byte level)
    - inefficient to load or store datum that spans quad word boundaries
    - virtual memory management tricky if data can span two pages

- Alignment rules upheld by compiler / assembly programmer
  - location of each variable according to alignment rules
    - gaps (padding) inserted as needed
    - also within composite data structures (structs, arrays, unions)
  - knowledge of alignment necessary to understand assembly code

- More on alignment of composite data structures later

CSE 컴퓨터공학부 Department of Computer Science & Engineering

RISC-V Data Alignment Conventions

Type	64-bit RISC-V		32-bit RISC-V	
	Size	Alignment	Size	Alignment
`bool`	1	1	1	1
`char`	1	1	1	1
`short`	2	2	2	2
`int`	4	4	4	4
`long`	**8**	**8**	**4**	**4**
`long long` two difference	8	8	8	8
`__int128`	16	16	-	-
`void *`	**8**	**8**	**4**	**4**
`_Float16`	2	2	2	2
`float`	4	4	4	4
`double`	8	8	8	8
`long double`	16	16	16	16
`float _Complex`	8 X2	4	8 X2	4
`double _Complex`	16 X2	8	16 X2	8
`long double _Complex`	32 X2	16	32 X2	16

Composite Value

Data Alignment Example

- Print addresses of global variables
  - compile for rv64g
  - inspect assembly, disassembly
  - execute using spike simulator

performance
portability

```
$ riscv64-unknown-elf-gcc -Wall -O2 -fno-common \
    -mabi=lp64d -march=rv64g \
    -o globals64.s -S globals.c

$ riscv64-unknown-elf-gcc -Wall -O2 -fno-common \
    -mabi=lp64d -march=rv64g \
    -o globals64.o -c globals64.s

$ riscv64-unknown-elf-gcc -Wall -O2 -fno-common \
    -mabi=lp64d -march=rv64g
    -o globals64 globals64.o
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

double d;
float f;
void *p;
#ifdef __SIZEOF_INT128__
__int128 i128;
#endif
long long ll;
long l;
char c2;
int i;
short s;
char c1;

#define addr(x) printf("  %-8s: %4zu %8p\n", \
                       #x, sizeof(x), &x)

int main(void) {
  addr(d);
  addr(f);
  addr(p);
#ifdef __SIZEOF_INT128__
  addr(i128);
#endif
  addr(ll);
  addr(l);
  addr(c2);
  addr(i);
  addr(s);
  addr(c1);

  return EXIT_SUCCESS;
}
```

globals.c

Data Alignment Example

```
$ spike pk globals64
bbl loader
  d        :   8  0x1f220
  f        :   4  0x1f218
  p        :   8  0x1f210
  i128     :  16  0x1f290
  ll       :   8  0x1f208
  l        :   8  0x1f200
  c2       :   1  0x1f1f8
  i        :   4  0x1f1f4
  s        :   2  0x1f1f2
  c1       :   1  0x1f1f0

$ riscv64-unknown-elf-objdump -dD globals64 | less
…
Disassembly of section .sbss:
…
000000000001f1f0 <c1>:
000000000001f1f2 <s>:
000000000001f1f4 <i>:
000000000001f1f8 <c2>:
000000000001f200 <l>:
000000000001f208 <ll>:
000000000001f210 <p>:
000000000001f218 <f>:
000000000001f220 <d>:
…
Disassembly of section .bss:
…
000000000001f290 <i128>:
…
```

Size address

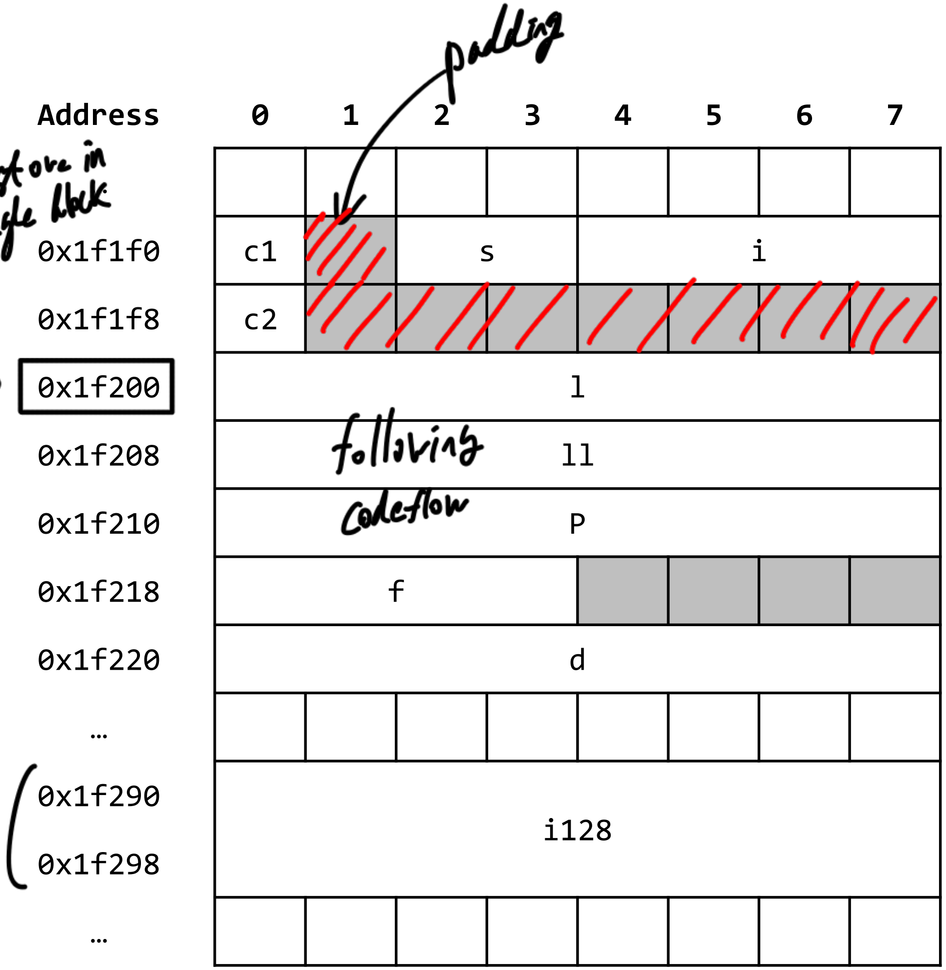
align order

Address gap must be Unit of 2

totally different section

prefer to store in single block

Special case?



- **Observations**
  - All variables properly aligned
  - RISC-V uses different sections for data
  - Layout in memory does not follow declaration order

Data Alignment Example (Local variable)

```
#include <stdio.h>
#include <stdlib.h>

#define addr(x) printf("  %-8s: %4lu %8p\n", \
                       #x, sizeof(x), &x)

int main(void) {
  double d;
  float f;
  // same entries and order as in globals.c
  short s;
  char c1;

  addr(d);
  …
  addr(c1);

  return EXIT_SUCCESS;
}
```

locals.c

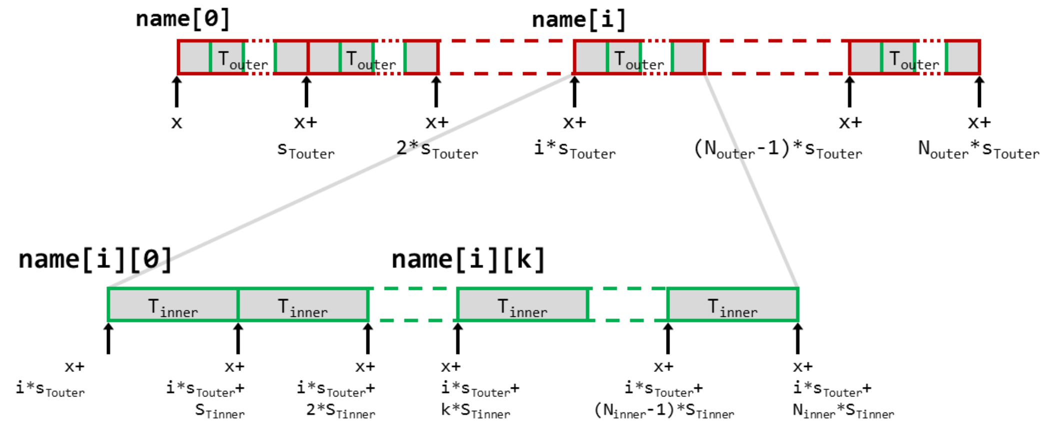
```
$ spike pk locals64
bbl loader
  d       :    8 0x3fff fffb10
  f       :    4 0x3ffffffb08
  p       :    8 0x3ffffffb18
  i128    :   16 0x3ffffffb30
  ll      :    8 0x3ffffffb20
  l       :    8 0x3ffffffb28
  c2      :    1 0x3ffffffb04
  i       :    4 0x3ffffffb0c
  s       :    2 0x3ffffffb06
  c1      :    1 0x3ffffffb05
```

minimize padding

Address	0	1	2	3	4	5	6	7
…								
0x3ffffffb00					c2	c1	s	
0x3ffffffb08	f				i			
0x3ffffffb10	d							
0x3ffffffb18	p							
0x3ffffffb20	ll							
0x3ffffffb28	L							
0x3ffffffb30	i128							
0x3ffffffb38								
…								

- Observations
  - All variables properly aligned
  - Local variables allocated on stack
  - Compiler reorders variables to minimize padding (minimize stack)

CSE 컴퓨터공학부
Department of Computer Science & Engineering


name[0]
name[i]
Touter
x
x+ sTouter
x+ 2*sTouter
x+ i*sTouter
x+ (Nouter-1)*sTouter
x+ Nouter*sTouter
name[i][0]
name[i][k]
Tinner
x+ i*sTouter
x+ i*sTouter+ STinner
x+ i*sTouter+ 2*STinner
x+ i*sTouter+ k*STinner
x+ i*sTouter+ (Ninner-1)*STinner
x+ i*sTouter+ Ninner*STinner

Arrays

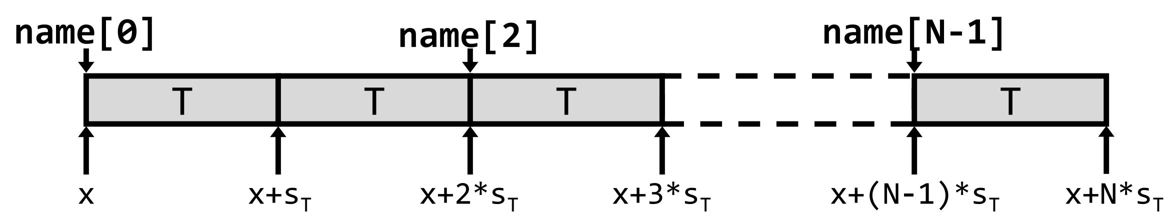
Arrays

- **Declaration**

  `<T> name[<N>]`

- **Size**
  - one element: s_T = sizeof(T)
  - entire array: $s_A = N * s_T$

- **Memory layout**



- **Address of i-th element**

  $$adr_i = name + i*s_T = \&name[i]$$

- **Alignment**
  - array alignment = alignment of base type T

Multidimensional Arrays

- **Formed when <type> is an array type**

outer = later dimension

$$\langle T_{outer}\rangle \ \texttt{name}[\langle N_{outer}\rangle]$$

$$\langle T_{outer}\rangle = \underbrace{\langle T_{inner}\rangle \ \ldots[\langle N_{inner}\rangle]}_{\text{inners}}$$

outest
inest
int arr[3][5]
⇒ arr[5] X3
⇒ (int X5) X3

combined notation

1 dim 2 dim

$$\langle T_{inner}\rangle \ \langle \texttt{name}\rangle[N_{outer}][N_{inner}]$$

- **Size**
  - one element: $s_{T_{outer}} = \texttt{sizeof}(T_{outer}) = (N_{inner} * s_{T_{inner}})$

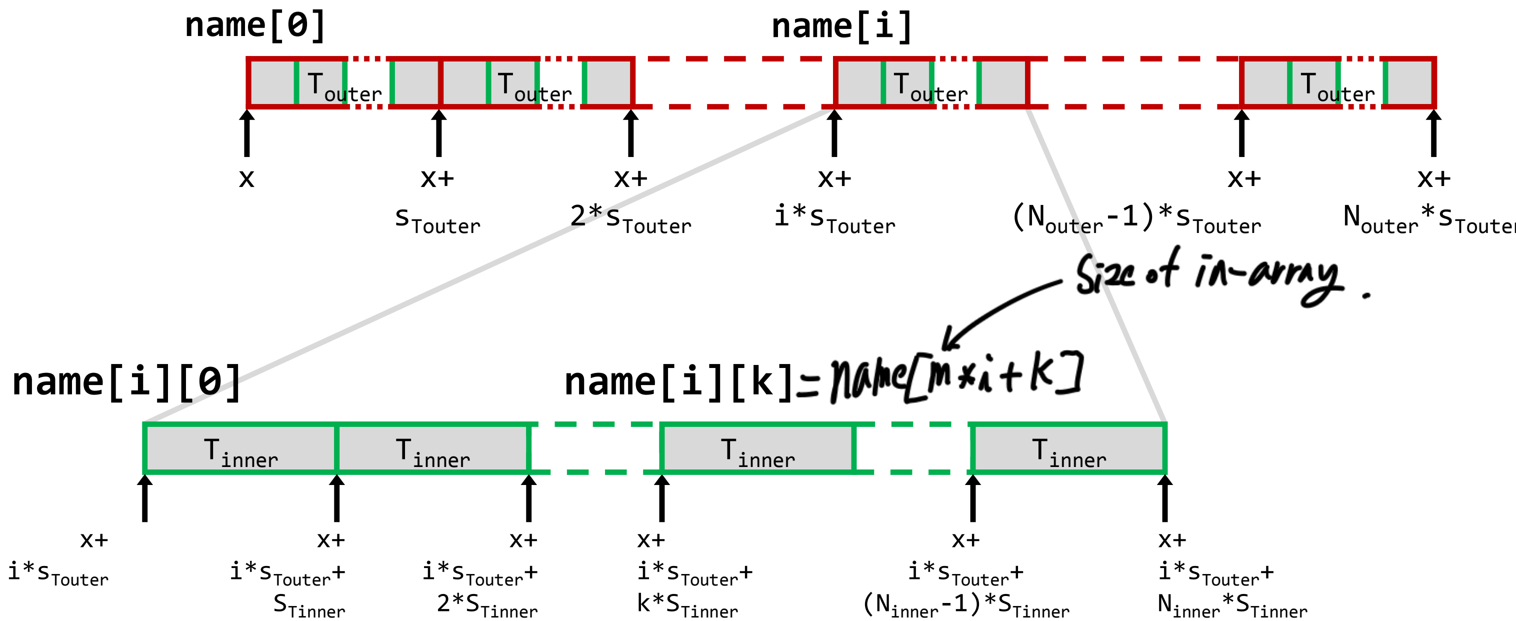
    inner array
  - entire array: $s_{A_{outer}} = N_{outer} * s_{T_{outer}} = N_{outer} * N_{inner} * s_{T_{inner}}$

Multidimensional Arrays

```
<T_outer> name[<N_outer>]
<T_outer> = <T_inner> …[<N_inner>]

<T_inner> <name>[N_outer][N_inner]
```

- **Memory layout ("row-major" layout)**

ex) name[n][m].



⇓

x+ $i*N_{inner}*S_{Tinner}+$ $k*S_{Tinner}$ ⇒ $\boxed{\text{name} + S_{Tinner}*(k + N_{inner}*i)}$

- **Address of element [i,k]**
  - note how N_{outer} is not needed to compute name[i][k]

Multidimensional Arrays

- **Generalization for n-dimensional array**

$$\langle T_{base}\rangle \ \langle name\rangle[N_{D_n}]\ldots[N_{D_2}][N_{D_1}]$$

- **Size**
  - entire array

$= \prod N_i \cdot \mathrm{Sizeof}(T)$

$$s_{D_n} = N_{D_n} * s_{T_{Dn-1}} = N_{D_n} * N_{D_{n-1}} * s_{T_{Dn-2}} = \ldots = \mathbf{N_{D_n} * N_{D_{n-1}} * \ldots * N_{D_1} * s_{T_{base}}}$$

  - subdimension k (n ≥ k ≥ 1)

$$s_{D_k} = N_{D_k} * s_{T_{Dk-1}} = \ldots = \mathbf{N_{D_k} * N_{D_{k-1}} * \ldots * N_{D_1} * s_{T_{base}}}$$

Multidimensional Arrays

- **Generalization for n-dimensional array**

$$\langle T_{base} \rangle\ \langle name \rangle [N_{D_n}]\ldots[N_{D_2}][N_{D_1}]$$

- **Address**
  - $name[i_{D_n}][i_{D_{n-1}}]\ldots[i_{D_2}][i_{D_1}]$

$$\mathbf{name + s_{T_{base}} * (i_{D_1} + N_{D_1} * (i_{D_2} + N_{D_2} * (\ldots + N_{D_{n-1}} * i_{D_n})\ldots))}$$

ex) $arr[N_3][N_2][N_1] \rightarrow arr[i][j][k] \Rightarrow *((int*)arr + k + N_1(j + N_2 \cdot i))$

  - again, note how $\mathbf{N_{D_n}}$ is not required for the address computation

N_3 is not exist!

  → this is why you can declare arrays with an open outermost dimension:

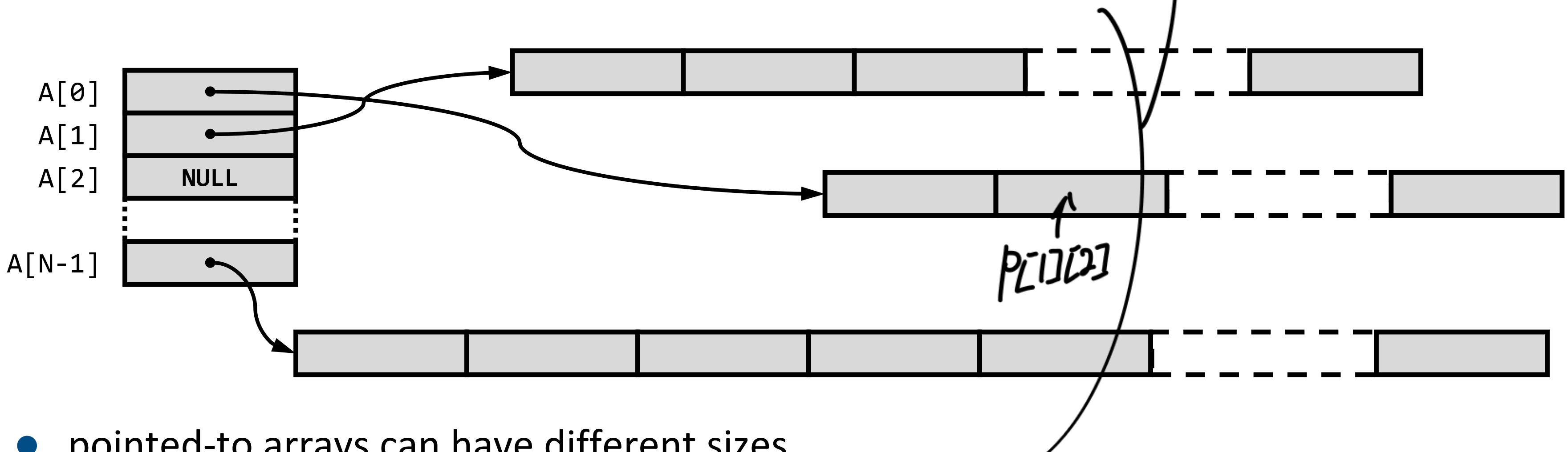
```
int A[][N][K];
```

Multilevel Arrays

- A multilevel array is an array of pointers to another array

  `int *A[N]`: Nm pointers

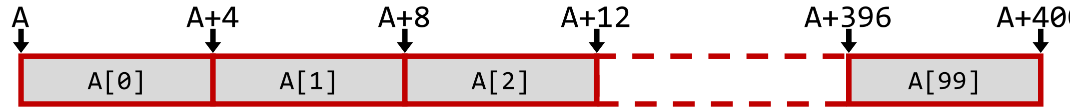
  each pointer can have different length



  - pointed-to arrays can have different sizes
  - elements in the pointer array can be NULL
  - requires one memory access per pointer indirection
  - both arrays, pointer array and pointed-to arrays can be multidimensional
  - address calculation separate
    pointer array: element = pointer; pointed-to array: any type

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Example: 1-d Array



One-dimensional array

```
#define N 100

int A[N];

int get(int i)
{
  return A[i];
}

int sum(void)
{
  int i, sum = 0;

  for (i=0; i<N; i++) {
    sum += A[i];
  }

  return sum;
}
```

array1.c

&A + i * sizeof(int)
= A + i*4 = A+i<<2

```
$ riscv64-unknown-elf-gcc \
    -mabi=lp64d -march=rv64g \
    -O2 -S array1.c
```

```
get:
    slli    a5,a0,2          a5=a0*4
    lui     a0,%hi(A)        ) a0=&A
    addi    a0,a0,%lo(A)
    add     a0,a0,a5         → a0 =&A[a0]
    lw      a0,0(a0)         → a0 = A[a0]
    ret

sum:
    lui     a5,%hi(A)        ) a5=&A
    addi    a5,a5,%lo(A)
    addi    a3,a5,400        → a3= &A[100] //last
    li      a0,0  reset
.L4:
    lw      a4,0(a5)         a4= A[i]
    addi    a5,a5,4          i=i+1
    addw    a0,a4,a0         a0= a4+a0 //sum
    bne     a5,a3,.L4        Check it
    ret
```

array1.s

sum compiled as:

```
int sum(void) {
  int *a5 = &A[0];
  int *a3 = &A[N];
  int a0 = 0;

  do {
    a4 = *a5; a5 += 4;
    sum += a4;
  while (a5 != a3);

  return a0;
}
```

Example: 2-d Array

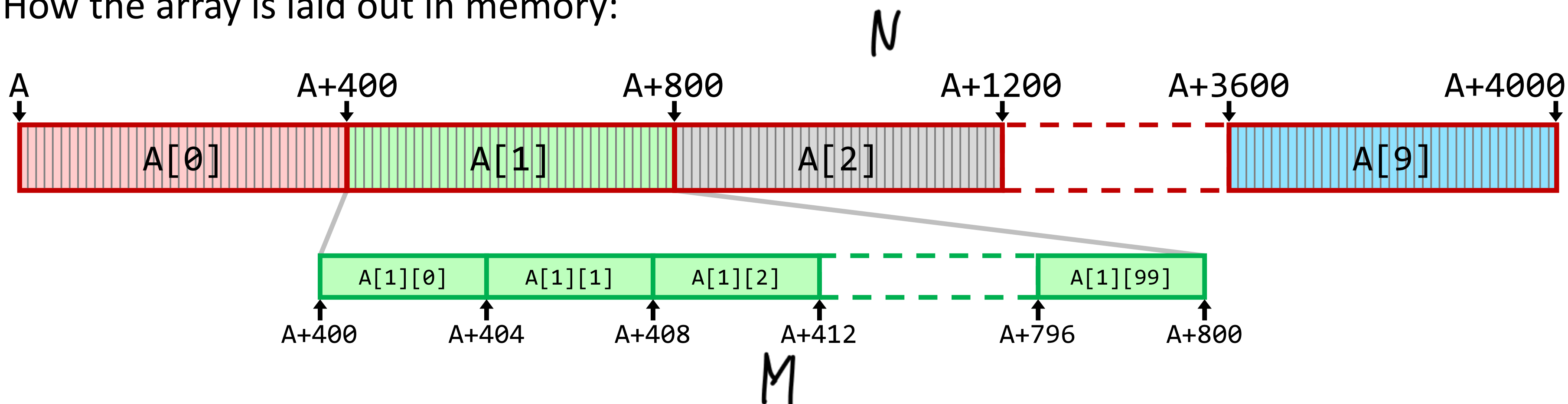
Two-dimensional array

```c
#define N 10
#define M 100

int A[N][M];
```
array2.c

How we imagine the array:

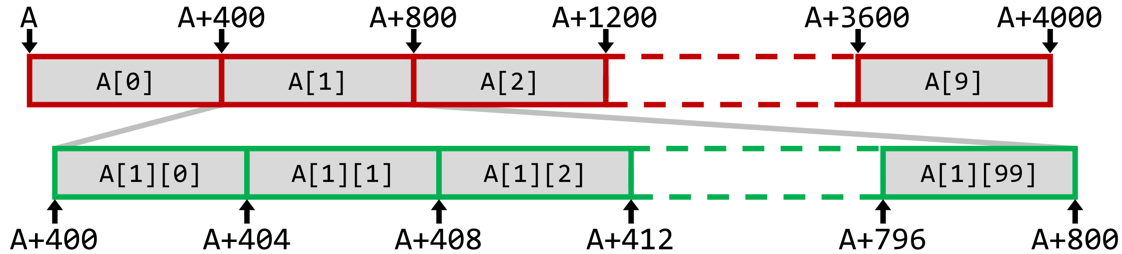
row/col	M: 0	1	2	…	99
N: 0	A[0][0]	A[0][1]	A[0][2]	…	A[0][99]
1	A[1][0]	A[1][1]	A[1][2]	…	A[1][99]
…	…	…	…	…	…
9	A[9][0]	A[9][1]	A[9][2]	…	A[9][99]

How the array is laid out in memory:



CSE 컴퓨터공학부 Department of Computer Science & Engineering

Example: 2-d Array



Two-dimensional array

```c
#define N 10
#define M 100

int A[N][M];

int get(int i, int j)
{
  return A[i][j];
}

int sum(void)
{
  int i, j, sum = 0;

  for (i=0; i<N; i++) {
    for (j=0; j<M; j++) {
      sum += A[i][j];
    }
  }

  return sum;
}
```

return A[i][j]; = &A[i+j*N]

= &A + sizeof(int)*(i+j*N)

>>2

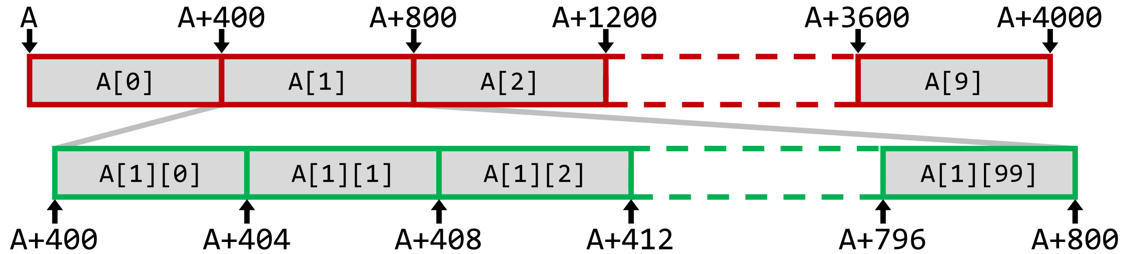
array2.c

```
get:
    li      a5,100        <- N
    mul     a0,a0,a5      <- j*N
    lui     a5,%hi(A)     ) a5= &A
    addi    a5,a5,%lo(A)  )
    add     a0,a0,a1      <- i+j*N
    slli    a0,a0,2       <- 4*(i+j*N)
    add     a0,a5,a0
    lw      a0,0(a0)
    ret

sum:
    lui     a2,%hi(A)     ) &A
    li      a5,4096       <- maximum imm value
    addi    a2,a2,%lo(A)  )
    addi    a5,a5,304     <- a5 = 4400
    addi    a3,a2,400     a3 = &A + sizeof(int) * 100
    li      a0,0
    add     a2,a2,a5
.L4:
    addi    a5,a3,-400
.L5:                      <- Summing
    lw      a4,0(a5)
    addi    a5,a5,4
    addw    a0,a4,a0
    bne     a5,a3,.L5     <- Loop
    addi    a3,a5,400     ) undo
    bne     a3,a2,.L4     -> next level
    ret
```

array2.s

$ riscv64-unknown-elf-gcc -mabi=lp64d -march=rv64g -O2 -S array2.c

Example: 2-d Array

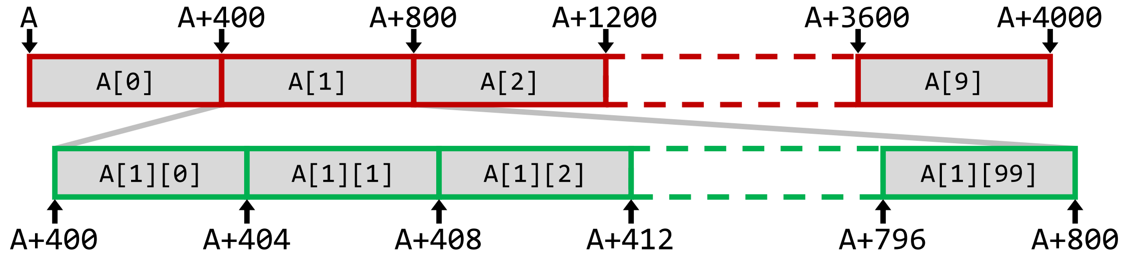


Two-dimensional array

```c
#define N 10
#define M 100

int A[N][M];

int get(int i, int j)
{
  return A[i][j];
}
```
array2.c

```
get:
  li    a5,100         # a5 = 100           // 100 elements in one row
  mul   a0,a0,a5       # a0 = i*100         // linear index of i-th row
  lui   a5,%hi(A)      # a5 =               // load address...
  addi  a5,a5,%lo(A)   #    = &A            //    ...of A
  add   a0,a0,a1       # a0 = i*100 + j     // index into A
  slli  a0,a0,2        # a0 = i*400 + j*4   // offset into A (as bytes)
  add   a0,a5,a0       # a0 = &A[i][j]      // address of element A[i][j]
  lw    a0,0(a0)       # a0 = A[i][j]       // load element into a0
  ret                  # return a0
```
array2.s

Example: 2-d Array



Two-dimensional array

array2.c

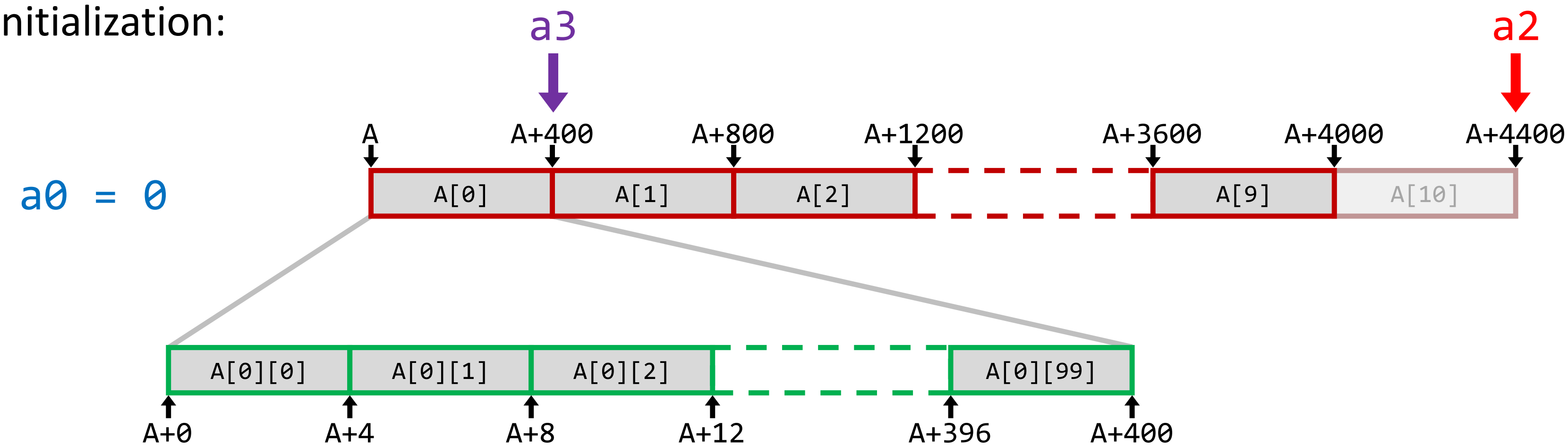
```c
#define N 10
#define M 100

int A[N][M];

int sum(void)
{
  int i, j, sum = 0;

  for (i=0; i<N; i++)
    for (j=0; j<M; j++)
      sum += A[i][j];

  return sum;
}
```

```asm
sum:
  lui    a2,%hi(A)       # a2 = &hi(A)
  li     a5,4096         # a5 = 4096
  addi   a2,a2,%lo(A)    # a2 = &A
  addi   a5,a5,304       # a5 = 4400
  addi   a3,a2,400       # a3 = &A + 400
  li     a0,0            # a0 = sum = 0
  add    a2,a2,a5        # a2 = &A + 4400

.L4:                     # i-loop
  addi   a5,a3,-400      # a5 = a3 – 400

.L5:                     # j-loop
  lw     a4,0(a5)        # a4 = mem[a5]
  addi   a5,a5,4         # a5 = a5+4
  addw   a0,a4,a0        # sum += a4
  bne    a5,a3,.L5       # a5 != a3 ? goto .L5

  addi   a3,a5,400       # a3 = a5 + 400
  bne    a3,a2,.L4       # a3 != a2 ? goto .L4

  ret                    # return sum
```

array2.s

Example: 2-d Array

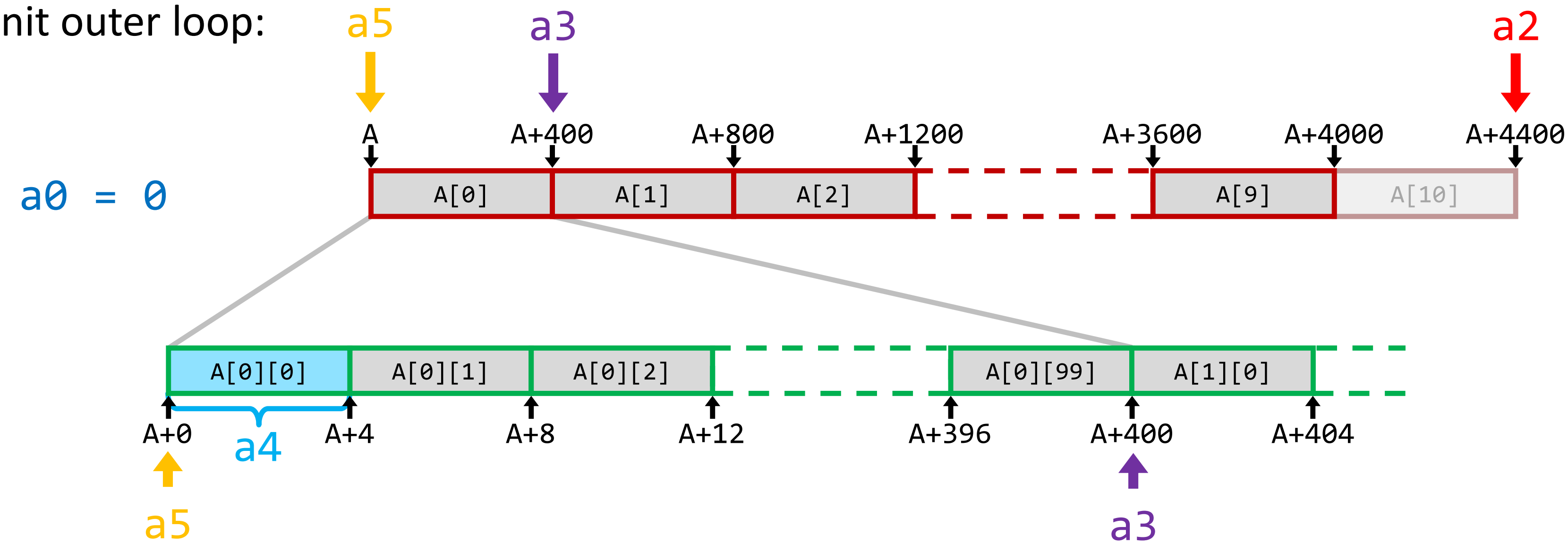
Initialization:




```
sum:
  lui    a2,%hi(A)      # a2 = &hi(A)
  li     a5,4096        # a5 = 4096
  addi   a2,a2,%lo(A)   # a2 = &A
  addi   a5,a5,304      # a5 = 4400
  addi   a3,a2,400      # a3 = &A + 400
  li     a0,0           # a0 = sum = 0
  add    a2,a2,a5       # a2 = &A + 4400
  ...
```

Example: 2-d Array

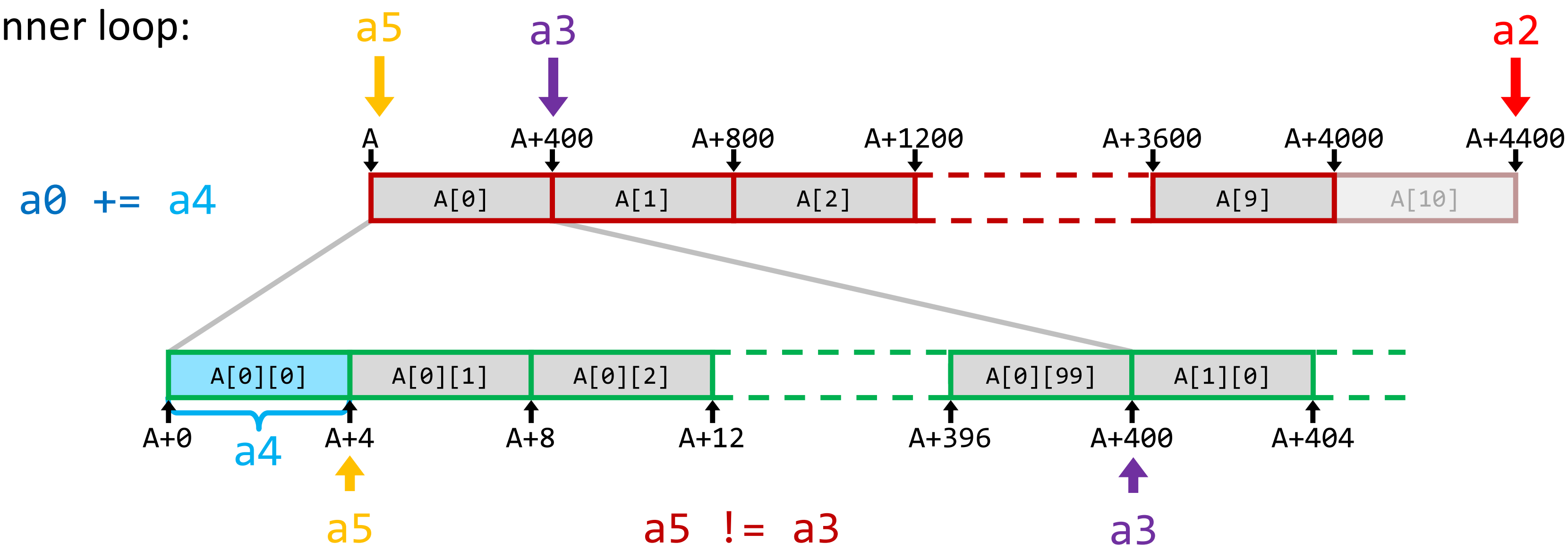
Init outer loop:




```
.L4:                      # i-loop
  addi   a5,a3,-400       # a5 = a3 - 400

.L5:                      # j-loop
  lw     a4,0(a5)         # a4 = mem[a5]
  addi   a5,a5,4          # a5 = a5+4
  addw   a0,a4,a0         # sum += a4
  bne    a5,a3,.L5        # a5 != a3 ? goto .L5

  addi   a3,a5,400        # a3 = a5 + 400
  bne    a3,a2,.L4        # a3 != a2 ? goto .L4
```

Example: 2-d Array

Inner loop:

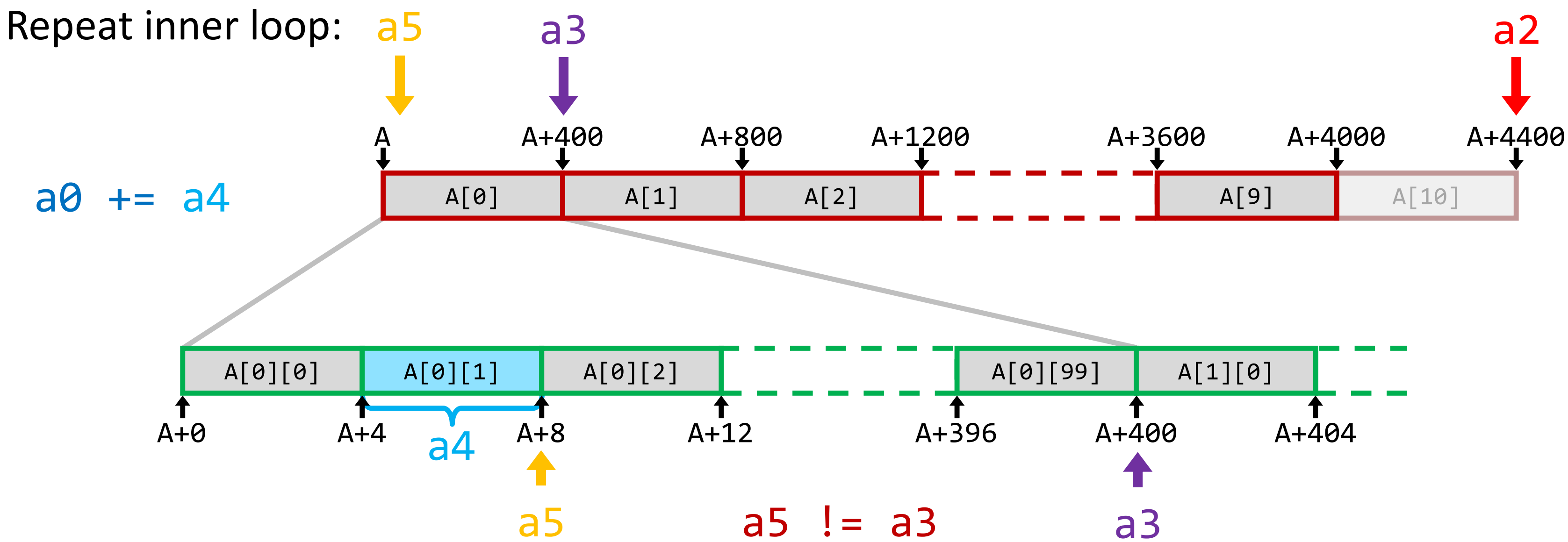



```
.L4:                       # i-loop
  addi   a5,a3,-400        # a5 = a3 – 400

.L5:                       # j-loop
  lw     a4,0(a5)          # a4 = mem[a5]
  addi   a5,a5,4           # a5 = a5+4
  addw   a0,a4,a0          # sum += a4
  bne    a5,a3,.L5         # a5 != a3 ? goto .L5

  addi   a3,a5,400         # a3 = a5 + 400
  bne    a3,a2,.L4         # a3 != a2 ? goto .L4
```

(check end of in array)

(check end of all array)

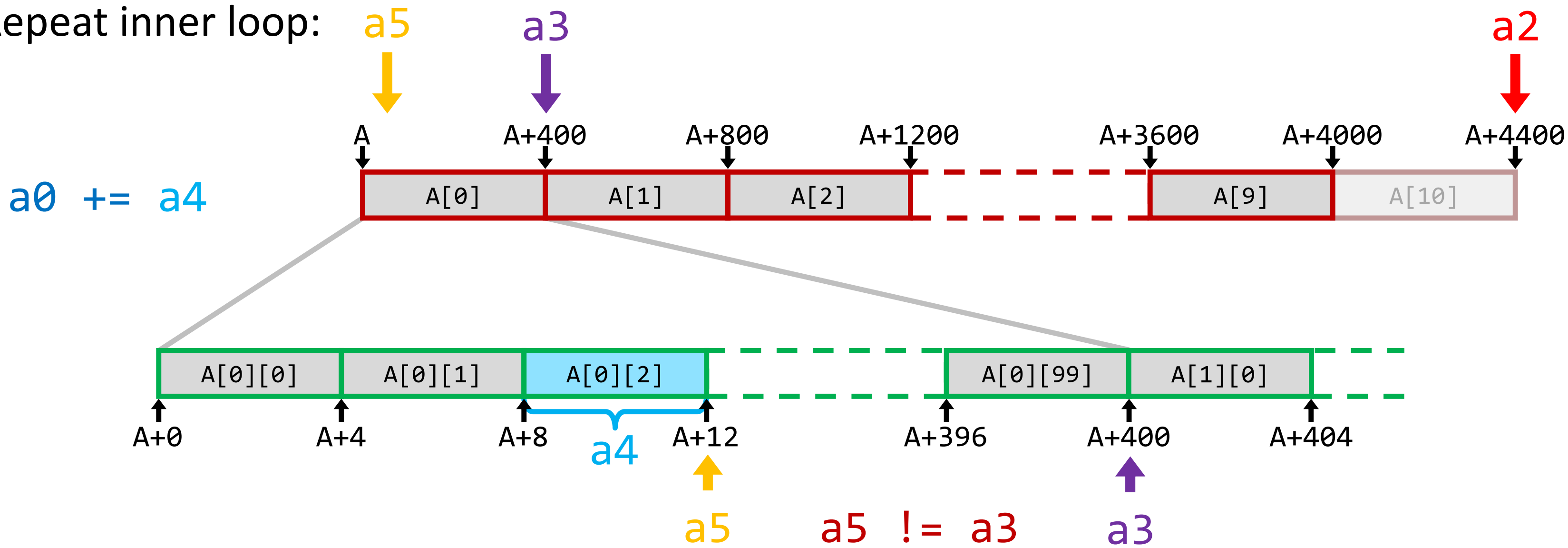
Example: 2-d Array




```
.L4:                    # i-loop
  addi   a5,a3,-400     # a5 = a3 – 400

.L5:                    # j-loop
  lw     a4,0(a5)       # a4 = mem[a5]
  addi   a5,a5,4        # a5 = a5+4
  addw   a0,a4,a0       # sum += a4
  bne    a5,a3,.L5      # a5 != a3 ? goto .L5

  addi   a3,a5,400      # a3 = a5 + 400
  bne    a3,a2,.L4      # a3 != a2 ? goto .L4
```

Example: 2-d Array

Repeat inner loop:

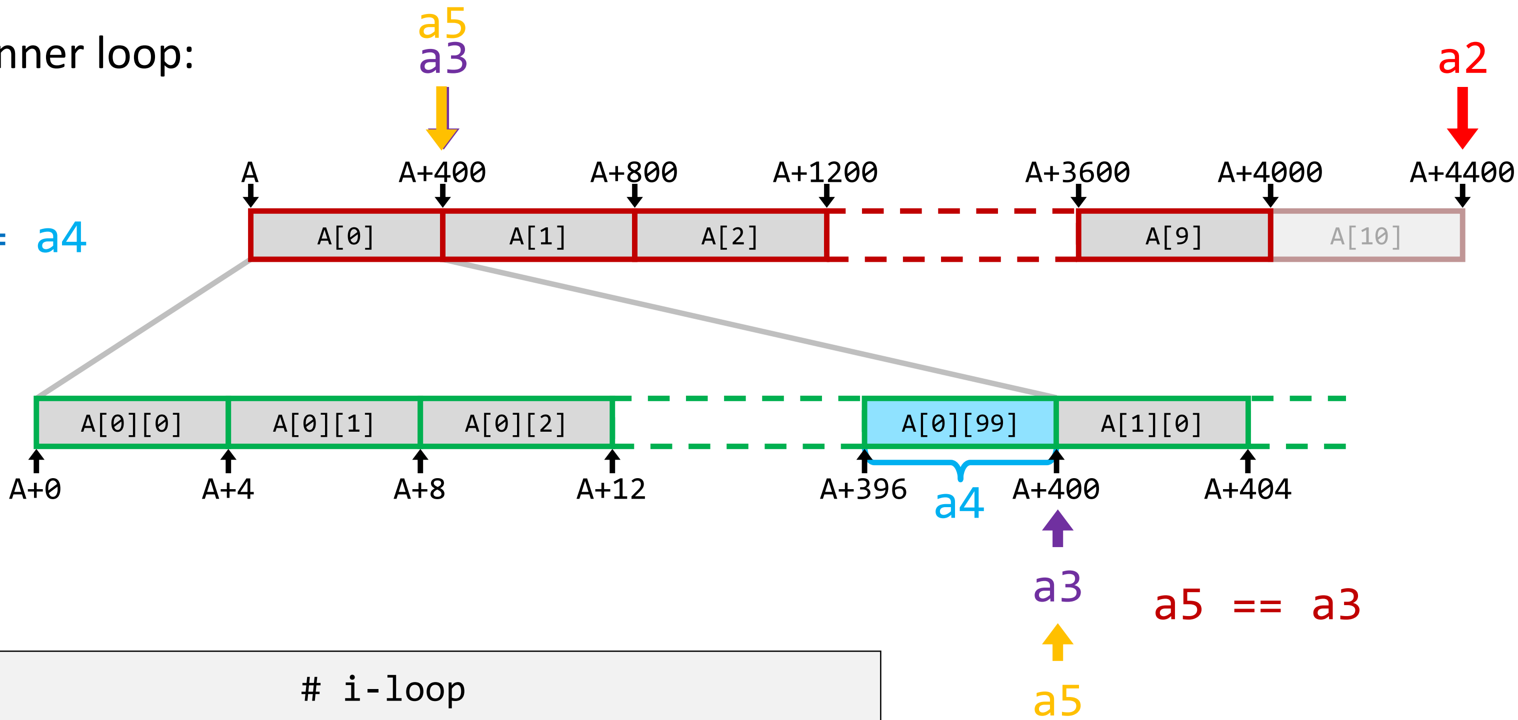



```
.L4:                    # i-loop
  addi  a5,a3,-400      # a5 = a3 – 400

.L5:                    # j-loop
  lw    a4,0(a5)        # a4 = mem[a5]
  addi  a5,a5,4         # a5 = a5+4
  addw  a0,a4,a0        # sum += a4
  bne   a5,a3,.L5       # a5 != a3 ? goto .L5

  addi  a3,a5,400       # a3 = a5 + 400
  bne   a3,a2,.L4       # a3 != a2 ? goto .L4
```

Example: 2-d Array

End of inner loop:

a0 += a4

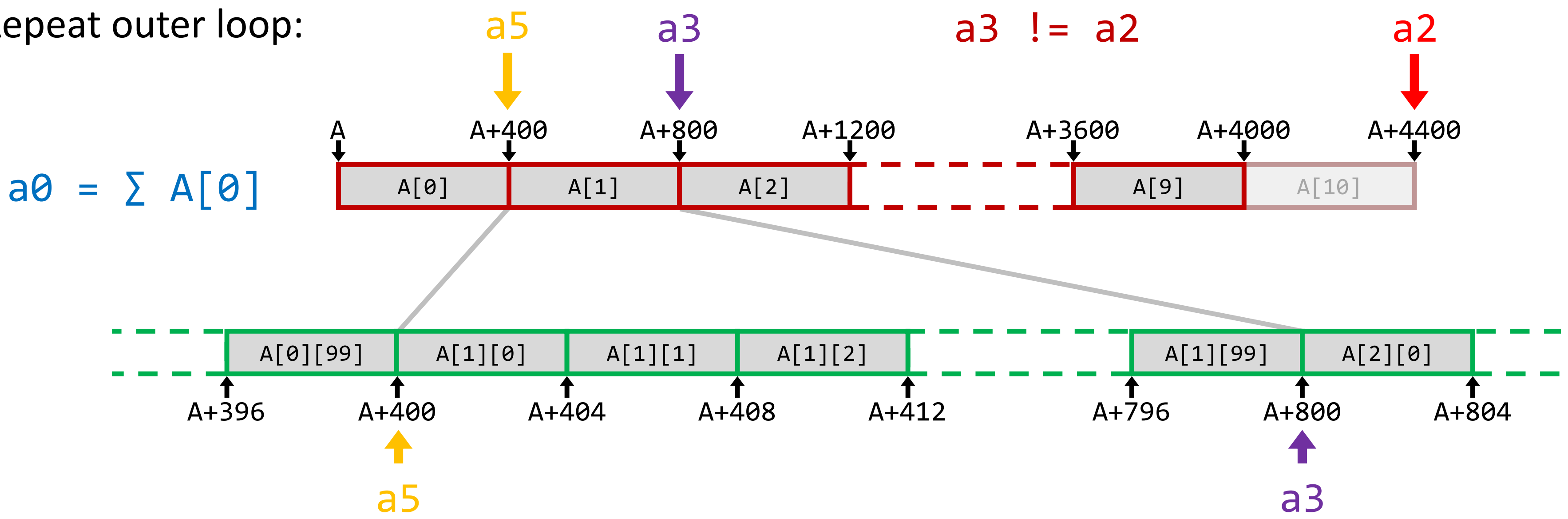



```
.L4:                    # i-loop
  addi  a5,a3,-400      # a5 = a3 – 400

.L5:                    # j-loop
  lw    a4,0(a5)        # a4 = mem[a5]
  addi  a5,a5,4         # a5 = a5+4
  addw  a0,a4,a0        # sum += a4
  bne   a5,a3,.L5       # a5 != a3 ? goto .L5

  addi  a3,a5,400       # a3 = a5 + 400
  bne   a3,a2,.L4       # a3 != a2 ? goto .L4
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering

Example: 2-d Array

Repeat outer loop:

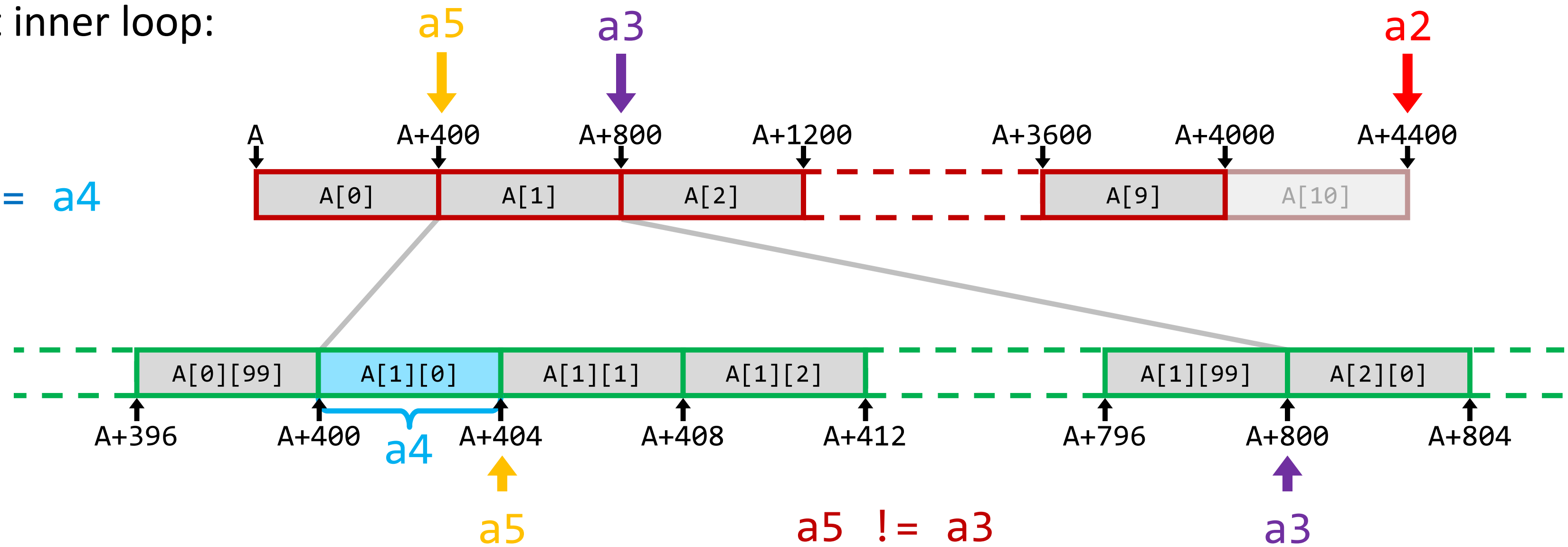



```
.L4:                    # i-loop
  addi  a5,a3,-400      # a5 = a3 – 400

.L5:                    # j-loop
  lw    a4,0(a5)        # a4 = mem[a5]
  addi  a5,a5,4         # a5 = a5+4
  addw  a0,a4,a0        # sum += a4
  bne   a5,a3,.L5       # a5 != a3 ? goto .L5

  addi  a3,a5,400       # a3 = a5 + 400
  bne   a3,a2,.L4       # a3 != a2 ? goto .L4
```

Observation: the RISC-V compiler is not omniscient!

Example: 2-d Array

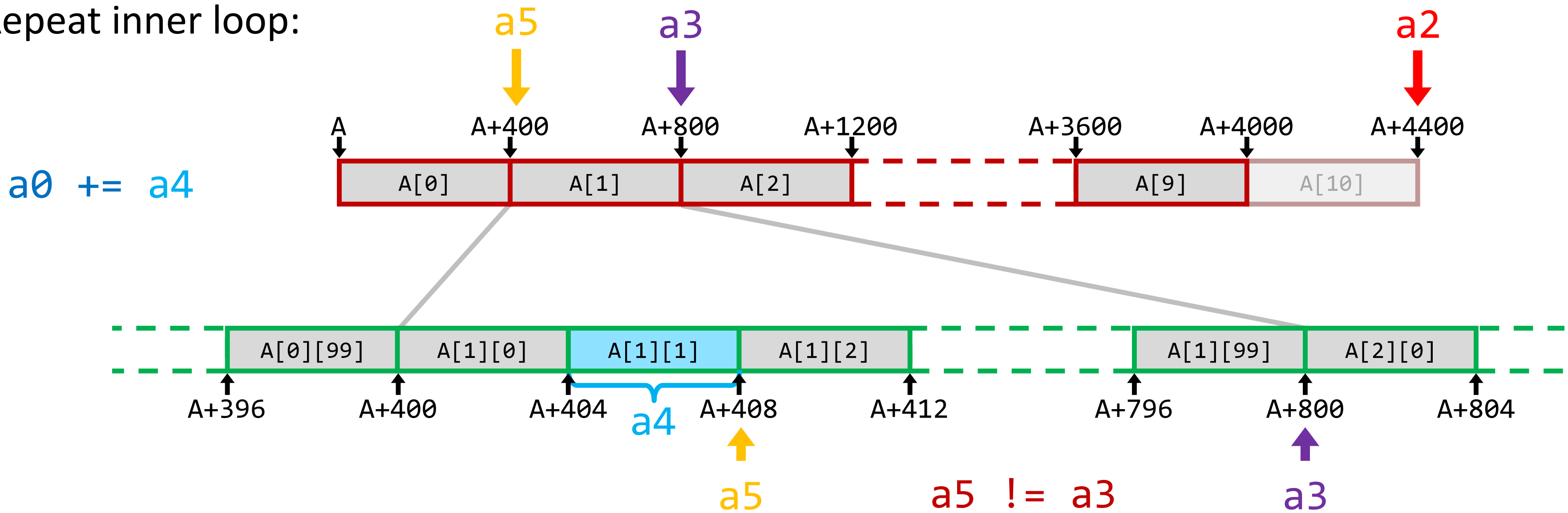
Repeat inner loop:




```
.L4:                    # i-loop
  addi   a5,a3,-400     # a5 = a3 – 400

.L5:                    # j-loop
  lw     a4,0(a5)       # a4 = mem[a5]
  addi   a5,a5,4        # a5 = a5+4
  addw   a0,a4,a0       # sum += a4
  bne    a5,a3,.L5      # a5 != a3 ? goto .L5

  addi   a3,a5,400      # a3 = a5 + 400
  bne    a3,a2,.L4      # a3 != a2 ? goto .L4
```

Example: 2-d Array

Repeat inner loop:

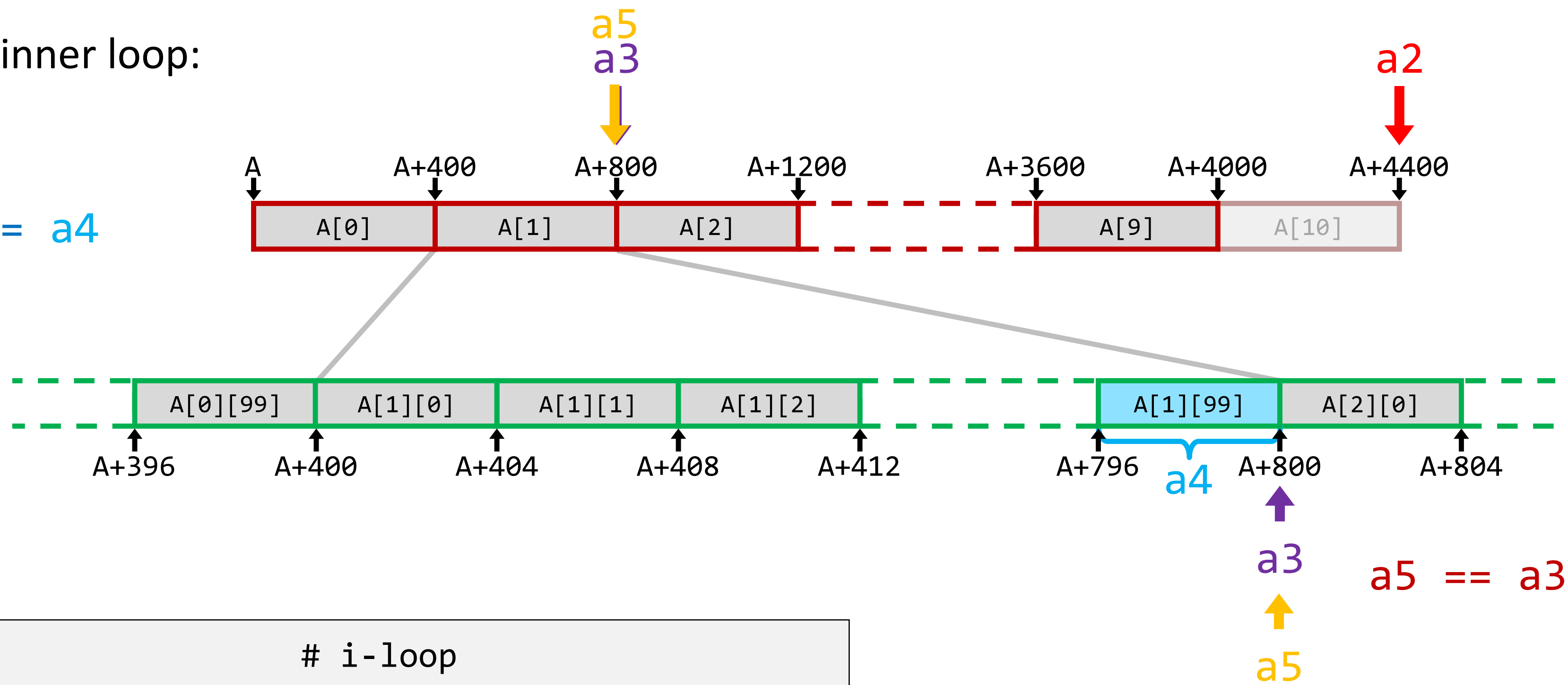



```
.L4:                    # i-loop
  addi  a5,a3,-400      # a5 = a3 – 400

.L5:                    # j-loop
  lw    a4,0(a5)        # a4 = mem[a5]
  addi  a5,a5,4         # a5 = a5+4
  addw  a0,a4,a0        # sum += a4
  bne   a5,a3,.L5       # a5 != a3 ? goto .L5

  addi  a3,a5,400       # a3 = a5 + 400
  bne   a3,a2,.L4       # a3 != a2 ? goto .L4
```

Example: 2-d Array

End of inner loop:

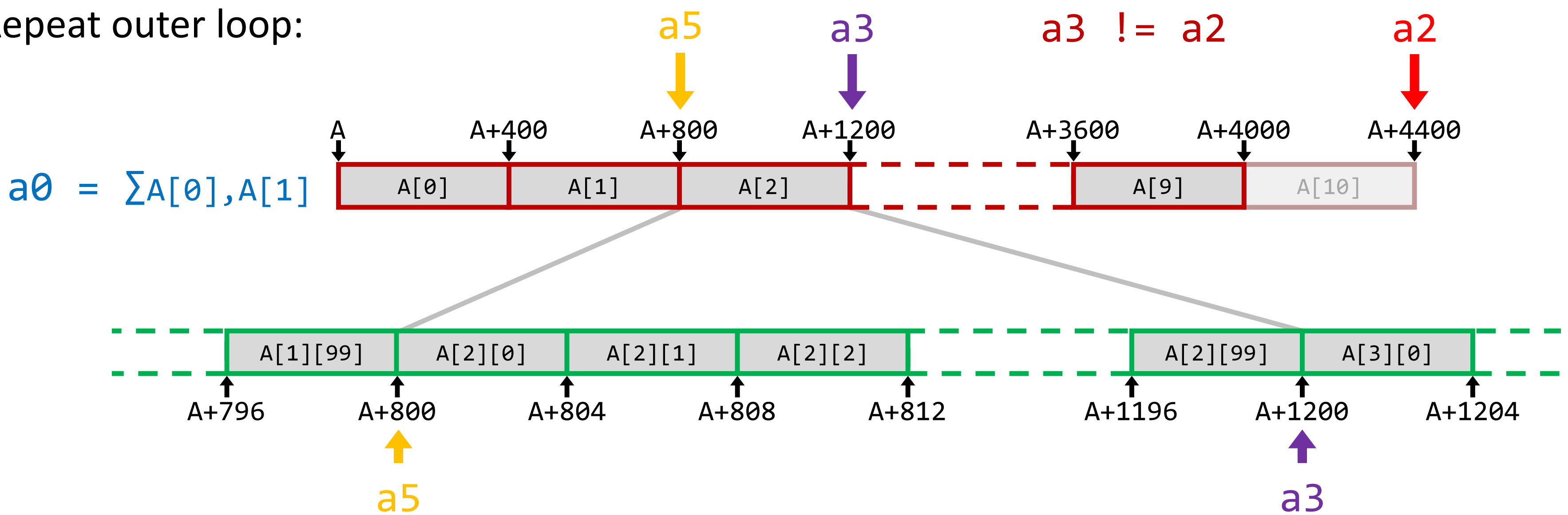
a0 += a4




```
.L4:                   # i-loop
  addi  a5,a3,-400     # a5 = a3 – 400

.L5:                   # j-loop
  lw    a4,0(a5)       # a4 = mem[a5]
  addi  a5,a5,4        # a5 = a5+4
  addw  a0,a4,a0       # sum += a4
  bne   a5,a3,.L5      # a5 != a3 ? goto .L5

  addi  a3,a5,400      # a3 = a5 + 400
  bne   a3,a2,.L4      # a3 != a2 ? goto .L4
```

Example: 2-d Array

Repeat outer loop:

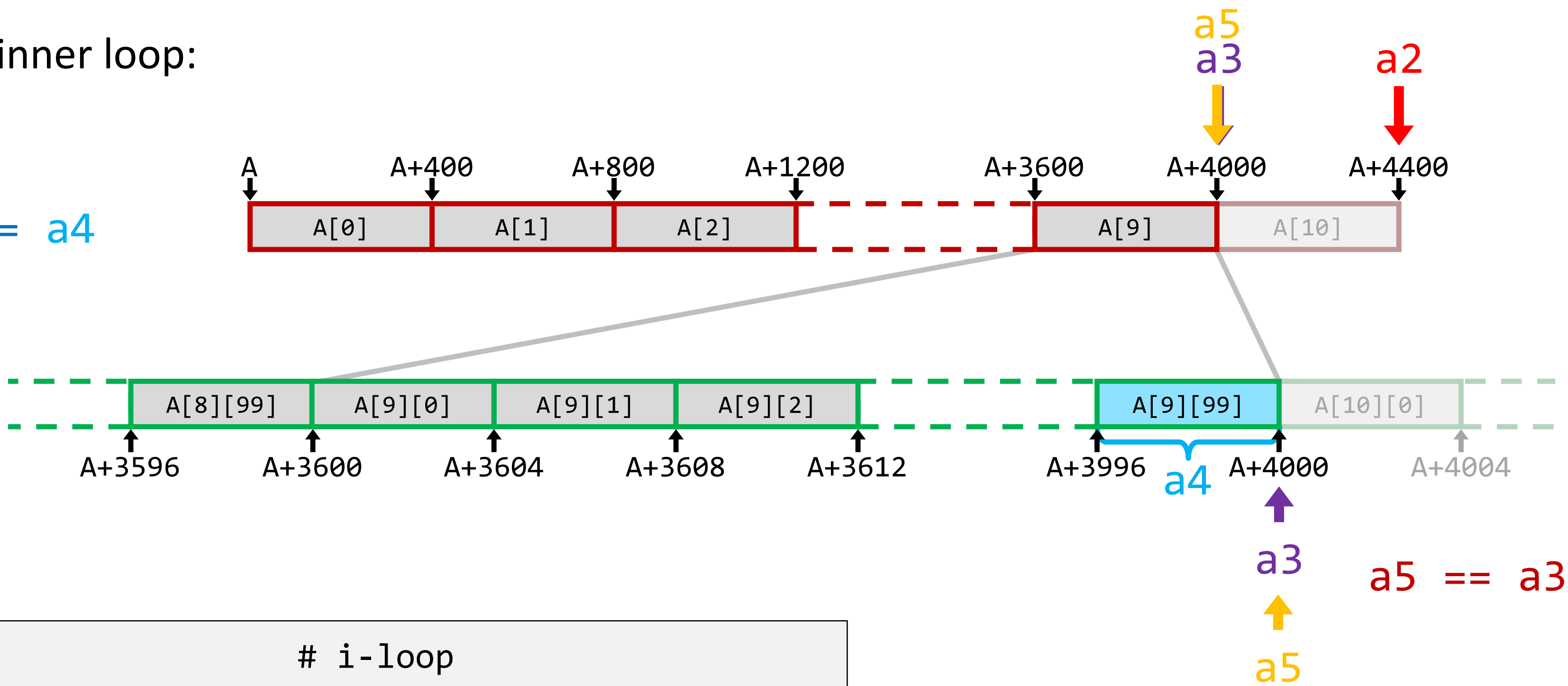



```
.L4:                    # i-loop
  addi  a5,a3,-400      # a5 = a3 – 400

.L5:                    # j-loop
  lw    a4,0(a5)        # a4 = mem[a5]
  addi  a5,a5,4         # a5 = a5+4
  addw  a0,a4,a0        # sum += a4
  bne   a5,a3,.L5       # a5 != a3 ? goto .L5

  addi  a3,a5,400       # a3 = a5 + 400
  bne   a3,a2,.L4       # a3 != a2 ? goto .L4
```

Example: 2-d Array

End of inner loop:

a0 += a4

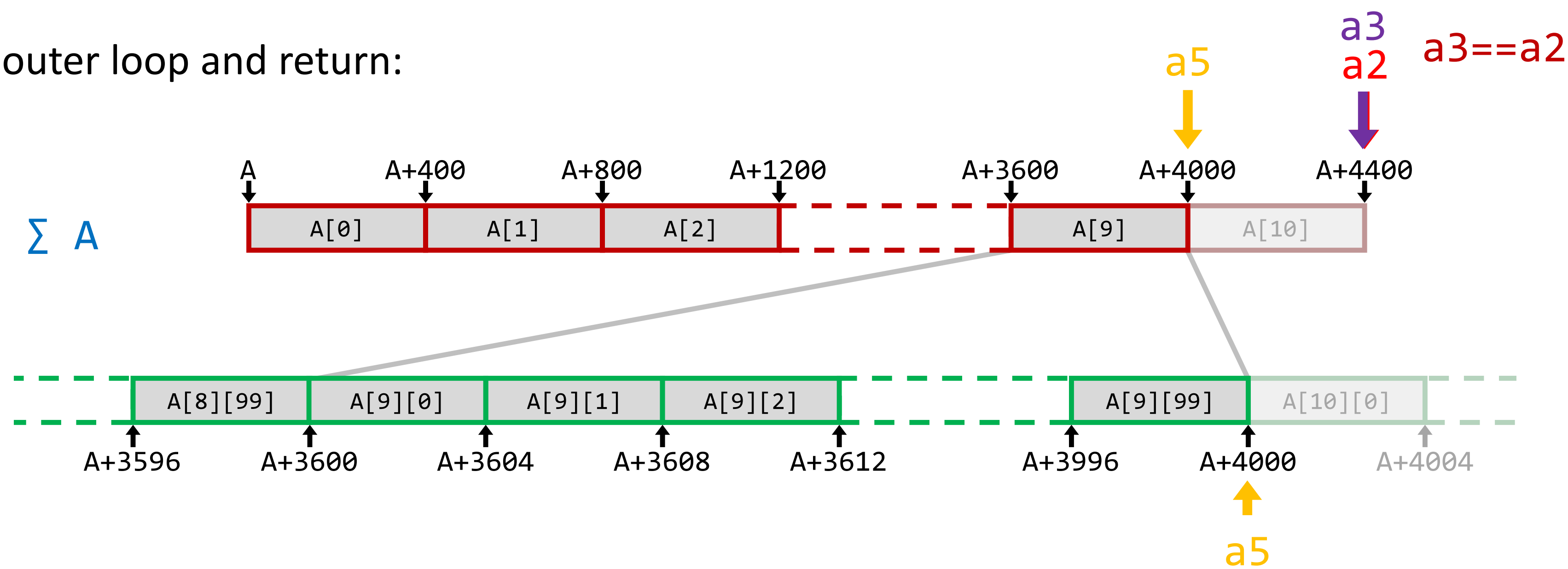



```
.L4:                    # i-loop
  addi  a5,a3,-400      # a5 = a3 – 400

.L5:                    # j-loop
  lw    a4,0(a5)        # a4 = mem[a5]
  addi  a5,a5,4         # a5 = a5+4
  addw  a0,a4,a0        # sum += a4
  bne   a5,a3,.L5       # a5 != a3 ? goto .L5

  addi  a3,a5,400       # a3 = a5 + 400
  bne   a3,a2,.L4       # a3 != a2 ? goto .L4
```

Example: 2-d Array

End of outer loop and return:

a0 = ∑ A




```
.L5:                  # j-loop
  lw     a4,0(a5)     # a4 = mem[a5]
  addi   a5,a5,4      # a5 = a5+4
  addw   a0,a4,a0     # sum += a4
  bne    a5,a3,.L5    # a5 != a3 ? goto .L5

  addi   a3,a5,400    # a3 = a5 + 400
  bne    a3,a2,.L4    # a3 != a2 ? goto .L4

  ret                 # return sum
```

a0 = ∑ A

Example: 2-d Array Column Sum

Two-dimensional array

```c
#define N 10
#define M 100

int A[N][M];

int colsum(int c)
{
  int i, sum = 0;

  for (i=0; i<N; i++) {
    sum += A[i][c];
  }

  return sum;
}
```

array3.c

Logical computation:

M

row/col	0	1	2	…	99
0	A[0][0]	A[0][1]	A[0][2]	…	A[0][99]
1	A[1][0]	A[1][1]	A[1][2]	…	A[1][99]
…	…	…	…	…	…
9	A[9][0]	A[9][1]	A[9][2]	…	A[9][99]

N

Σ

Example: 2-d Array Column Sum

Two-dimensional array

```
#define N 10
#define M 100

int A[N][M];

int colsum(int c)
{
  int i, sum = 0;

  for (i=0; i<N; i++) {
    sum += A[i][c];
  }

  return sum;
}
```
array3.c

Logical computation:

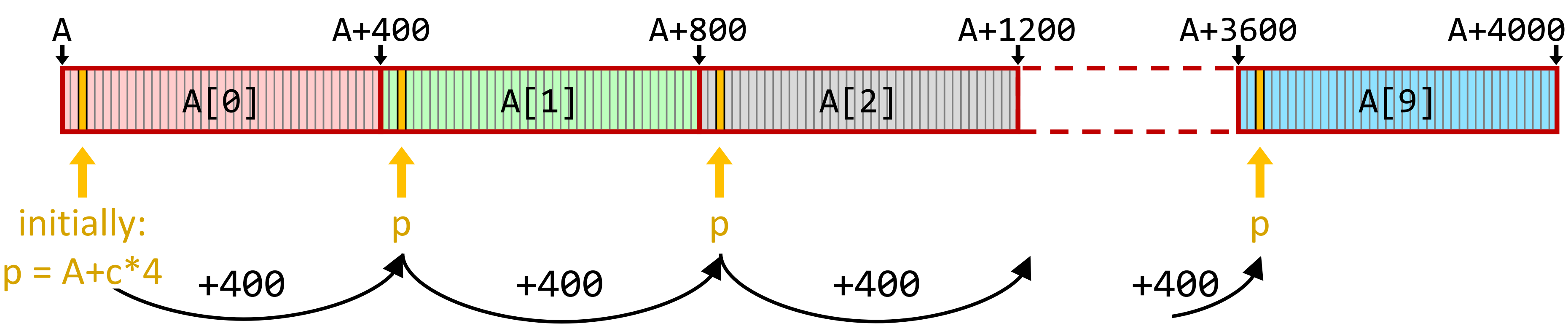
M

row/col	0	1	2	…	99
0	A[0][0]	A[0][1]	A[0][2]	…	A[0][99]
1	A[1][0]	A[1][1]	A[1][2]	…	A[1][99]
…	…	…	…	…	…
9	A[9][0]	A[9][1]	A[9][2]	…	A[9][99]

N

Σ

Physical computation:



Example: 2-d Array Column Sum

Two-dimensional array

```
#define N 10
#define M 100

int A[N][M];

int colsum(int c)
{
  int i, sum = 0;

  for (i=0; i<N; i++) {
    sum += A[i][c];
  }

  return sum;
}
```

array3.c

```
$ riscv64-unknown-elf-gcc \
    -mabi=lp64d -march=rv64g \
    -O2 -S array3.c
```

```
colsum:
    lui     a5,%hi(A)
    li      a3,4096
    addi    a5,a5,%lo(A)    # a5 = &A
    addi    a3,a3,-96       # a3 = 4000
    slli    a0,a0,2         # a0 = c*4
    add     a3,a5,a3        # a3 = &A + 4000
    add     a3,a3,a0        # a3 = &A + 4000 + c*4 //last addr
    add     a5,a0,a5        # a5 = &A + c*4
    li      a0,0            # a0 = 0
.L2:
    lw      a4,0(a5)        # a4 = mem[a5]
    addi    a5,a5,400       # a5 = a5 + 400
    addw    a0,a4,a0        # a0 = a0 + a4
    bne     a5,a3,.L2       # a5 != a3 ? goto .L2
    ret
```

array3.s

colsum compiled as:

```
int sum(void) {
  int *a5 = &A + c;      // C automatically multiplies c by 4
  int *a3 = &A[N] + c;   // idem
  int a0 = 0;

  do {
    a4 = *a5; a5 += 400;
    sum += a4;
  while (a5 != a3);

  return a0;
}
```

Example: Multidimensional vs. Multilevel Array

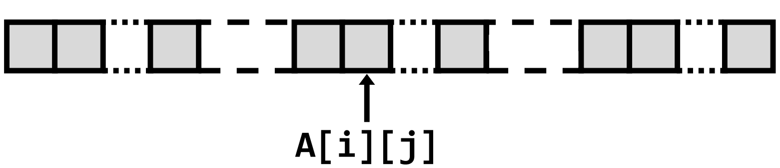
Multidimensional (nested) array

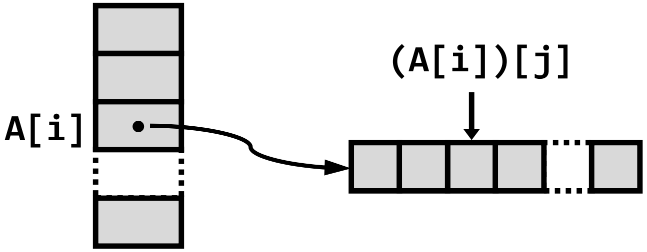
```
#define N 10
#define M 100

int A[N][M];

int get(int i, int j)
{
  return A[i][j];
}
```
array2.c

Multi-level array

```
#define N 10


int *A[N];

int get(int i, int j)
{
  return A[i][j];
}
```
array4.c

C code for get() looks identical, but access completely different!





MEM[A+4*(j + M*i)]

One memory access

MEM[MEM[A+4*i] + 4*j]

Multiple memory accesses

Example: Multidimensional vs. Multilevel Array

Multidimensional (nested) array

```c
#define N 10
#define M 100

int A[N][M];

int get(int i, int j)
{
  return A[i][j];
}
```
array2.c

```
get:
    li      a5,100
    mul     a0,a0,a5
    lui     a5,%hi(A)
    addi    a5,a5,%lo(A)
    add     a0,a0,a1
    slli    a0,a0,2
    add     a0,a5,a0
    lw      a0,0(a0)   # load element
    ret
```
array2.s

`MEM[ A+4*(j + M*i) ]`

Multi-level array

```c
#define N 10


int *A[N];

int get(int i, int j)
{
  return A[i][j];
}
```
array4.c

```
get:
    slli    a0,a0,3
    lui     a5,%hi(A)
    addi    a5,a5,%lo(A)
    add     a5,a5,a0
    ld      a5,0(a5)   # load pointer
    slli    a1,a1,2
    add     a5,a5,a1
    lw      a0,0(a5)   # load element
    ret
```
array4.s

two load!

`MEM[ MEM[ A+8*i ] + 4*j ]`

Example: Multidimensional vs. Multilevel Array

Multidimensional (nested) array

```
#define N 5

int A[N][N][N];
              a0        a1        a2
int get(int i, int j, int k)
{
  return A[i][j][k];
}
```

array5.c

4* (k + 5(j + 5*i))

```
slli  a5, a0, 2     ) 5*i
add   a5, a0, a5    )
add   a1, a0, a1
slli  a5, a1, 2
add   a5, a1, a0
add   a2, a5, a2
slli  a2, a2, 2
```

faster!

```
get:
  slli   a5,a0,1        # a5 = i*2
  add    a5,a5,a0       # a5 = i*3
  slli   a4,a1,2        # a4 = j*4
  slli   a5,a5,3        # a5 = i*24
  add    a5,a5,a0       # a5 = i*25
  add    a1,a4,a1       # a1 = j*5
  add    a5,a5,a1       # a5 = i*25 + j*5
  add    a5,a5,a2       # a5 = i*25 + j*5 + k
  lui    a2,%hi(A)      # a2 =
  addi   a2,a2,%lo(A)   #    = &A
  slli   a5,a5,2        # a5 = 4*(25i+5j+k)
  add    a5,a2,a5       # a5 = &A[i][j][k]
  lw     a0,0(a5)       # load MEM[a5]
  ret
```

array5.s

MEM[A+4*(k + 5*(j + 5*i))]

Multi-level array

```
#define N 10

int **A[N];

int get(int i, int j, int k)
{
  return A[i][j][k];
}
```

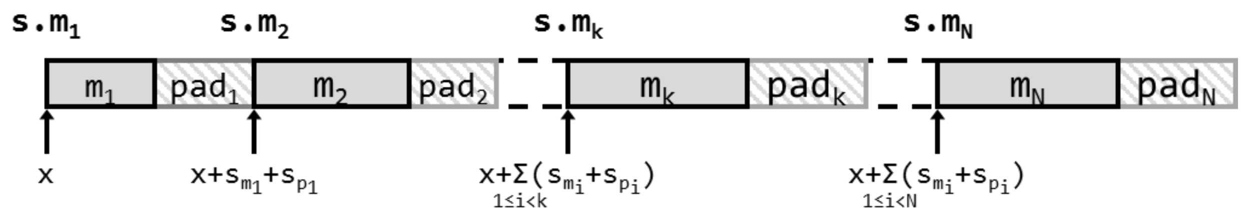
array6.c

```
get:
  lui    a5,%hi(A)      # a5 =
  slli   a0,a0,3        # a0 = i*8
  addi   a5,a5,%lo(A)   #    = &A
  add    a5,a5,a0       # a5 = &(A + i*8)
  ld     a5,0(a5)       # a5 = load A[i]
  slli   a1,a1,3        # a1 = j*8
  slli   a2,a2,2        # a2 = k*4
  add    a5,a5,a1       # a5 = &(A[i] + j*8)
  ld     a5,0(a5)       # a5 = load A[i][j]
  add    a5,a5,a2       # a5 = &(a5 + k*4)
  lw     a0,0(a5)       # a0 = load MEM[a5]
  ret
```

multi load

array6.s

MEM[MEM[MEM[A+8*i] + 8*j] + 4*k]


s.m1
s.m2
s.mk
s.mN
m1
pad1
m2
pad2
mk
padk
mN
padN
x
x+sm1+sp1
x+Σ(smi+spi)
1≤i<k
x+Σ(smi+spi)
1≤i<N

Structures

Structures

Declaration

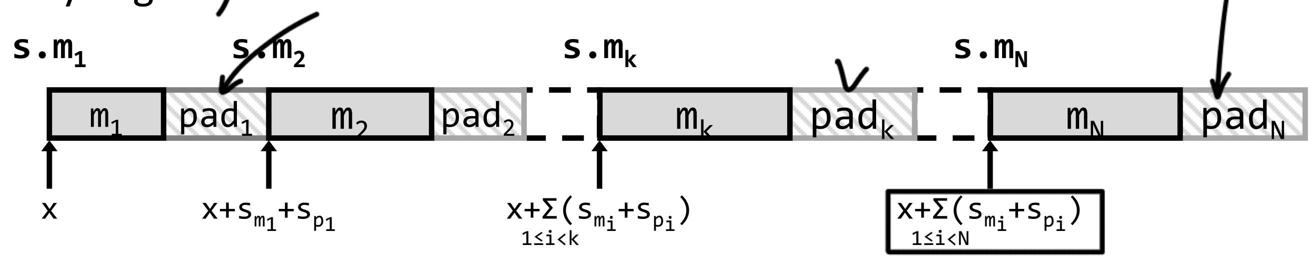
```
struct name {
  <T1> <m1>;
  <T2> <m2>;
  …
  <TN> <mN>;
};
```

in order (fixed)

end-padding for struct array case

Memory layout

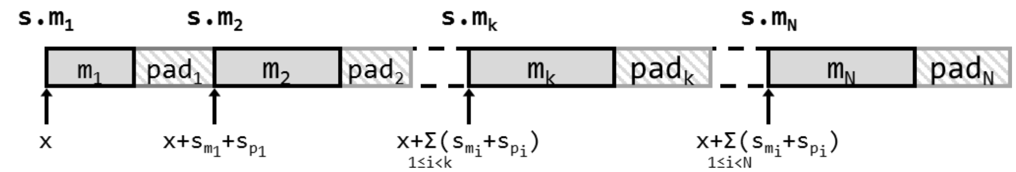
- consecutive memory region containing all members m_i (in-order, non-overlapping, and properly aligned)



Alignment

- struct alignment = maximum alignment requirement of any of its members
- member alignment = alignment requirement of member type
  - padding: "holes" in the memory layout to maintain alignment requirements (denoted $\mathtt{pad}_i$ above)

Structures



- **Address of k-th member**
  - start of struct plus sum of sizes of all 1..k-1 members and paddings

$$adr_{m_k} = x + \sum_{1 \le i < k} (s_{m_i} + s_{p_i})$$

- **Size of struct**

$$s_s = \sum_{1 \le i \le N} (s_{m_i} + s_{p_i})$$

  - the last padding (pad_N) is chosen such that the size of the struct is the next multiple of the biggest alignment requirement of any of its members
    - this comes in handy when declaring arrays of structs (all elements of the array will be automatically aligned)

Structure Layout & Alignment

Structure layout & alignment

```
struct s1 {
  char      c1;
  char      c2;
  int       i;
  short     s;
  long long l;
  float     f;
  double    d;
};
                  struct1.c
```

RV64 size	RV64 align
40	8
1	1
1	1
4	4
2	2
8	8
4	4
8	8

in-order

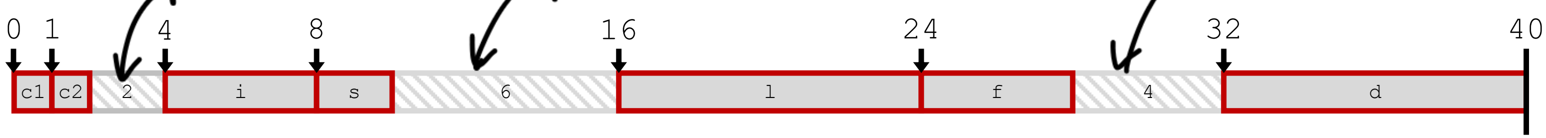
max(8) ⇒ end of struct's address start with Domain of 8

padding for int

padding for long

padding for double

RV64:



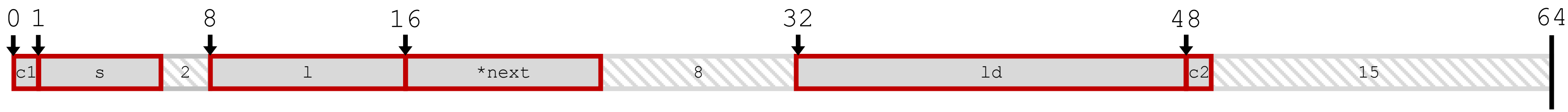
Structure Layout & Alignment

Structure layout & alignment

```
struct s2 {
  char        c1;
  char        s[5];
  long        l;
  struct s2   *next;
  long double ld;
  char        c2;
};                 struct2.c
```

	RV64	
	size	align
struct s2	64	16
c1	1	1
s[5]	5	1
l	8	8
*next	8	8
ld	16	16
c2	1	1

RV64:



Size & Alignment of Primitive Types

Finding out about sizes and alignments on your machine

```
#include <stdio.h>

#define SIZE(t) sizeof(t)
#define OFS(v) ((unsigned long)&v.data - (unsigned long)&v)
#define INFO(t) { struct { char dummy; t data; } s; \
                  printf("%-15s %2zu       %2lu\n", #t, SIZE(t), OFS(s)); }

void main(void)
{ printf("DATATYPE       SIZE   ALIGNMENT\n");
  INFO(char);
  INFO(short);
  INFO(int);
  INFO(long);
  INFO(long long);
  INFO(float);
  INFO(double);
  INFO(long double);
  INFO(void *);
}
```

size.c

Macro-expansion generates the following code pattern:

```
void main(void)
{ printf("DATATYPE       SIZE   ALIGNMENT\n");
  { struct { char dummy; char data; } s; printf("%-15s %2zu       %2lu\n",
    "char", sizeof(char), ((unsigned long)&s.data - (unsigned long)&s)); };
  { struct { char dummy; short data; } s; printf("%-15s %2zu       %2lu\n",
    "short", sizeof(short), ((unsigned long)&s.data - (unsigned long)&s)); };
  …
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering

Size & Alignment of Primitive Types

Finding out about sizes and alignments on your machine

```
$ risc64-unknown-elf-gcc -mabi=lp64d -march=rv64g -o size.rv64 size.c
$ spike pk size.rv64
DATATYPE        SIZE   ALIGNMENT
char               1           1
short              2           2
int                4           4
long               8           8
long long          8           8
float              4           4
double             8           8
long double       16          16
void *             8           8
$
```

RV64

```
$ gcc -m64 -o size.64 size.c
$ ./size.64
DATATYPE        SIZE   ALIGNMENT
char               1           1
short              2           2
int                4           4
long               8           8
long long          8           8
float              4           4
double             8           8
long double       16          16
void *             8           8
$
```

x86_64

```
$ gcc -m32 -o size.32 size.c
$ ./size.32
DATATYPE        SIZE   ALIGNMENT
char               1           1
short              2           2
int                4           4
long               4           4
long long          8           4
float              4           4
double             8           4
long double       12           4
void *             4           4
$
```

IA32

Size & Alignment of Structures

Finding out about structure offsets and paddings on your machine

```
#include <stdio.h>

struct s2 {
  char        c1;
  char        s[5];
  long        l;
  struct s2   *next;
  long double ld;
  char        c2;
};

#define SIZE(t) sizeof(t)
#define OFS(s,m) ((unsigned long)&s.m - (unsigned long)&s)
#define PAD(s,m,mn) (OFS(s,mn) - OFS(s,m) - sizeof(s.m))

void main(void)
{
  struct s2 s;

  printf("                   OFS  SIZE   PAD\n");
  printf("struct s2 {                 %2zu\n", SIZE(s));
  printf("  char        c1;        %2lu    %2zu    %2lu\n", OFS(s,c1),  SIZE(s.c1),  PAD(s,c1,  s));
  printf("  char        s[5];      %2lu    %2zu    %2lu\n", OFS(s,s),   SIZE(s.s),   PAD(s,s,   l));
  printf("  long        l;         %2lu    %2zu    %2lu\n", OFS(s,l),   SIZE(s.l),   PAD(s,l,   next));
  printf("  struct s2   *next;     %2lu    %2zu    %2lu\n", OFS(s,next),SIZE(s.next),PAD(s,next,ld));
  printf("  long double ld;        %2lu    %2zu    %2lu\n", OFS(s,ld),  SIZE(s.ld),  PAD(s,ld,  c2));
  printf("  char        c2;        %2lu    %2zu    %2lu\n", OFS(s,c2),  SIZE(s.c2),
                                                            SIZE(s)-OFS(s,c2)-SIZE(s.c2));
  printf("}\n");
}
```

struct2.c

Size & Alignment of Structures

Finding out about structure offsets and paddings on your machine

```
$ risc64-unknown-elf-gcc -mabi=lp64d -march=rv64g -o struct2.rv64 struct2.c
$ spike pk struct2.rv64
                        OFS  SIZE   PAD
struct s2 {                    64
  char        c1;         0     1     0
  char        s[5];       1     5     2
  long        l;          8     8     0
  struct s2   *next;     16     8     8
  long double ld;        32    16     0
  char        c2;        48     1    15
}
$                                                                      RV64
```

```
$ gcc -m64 -o struct2.64 struct2.c
$ ./struct2.64
                        OFS  SIZE   PAD
struct s2 {                    64
  char        c1;         0     1     0
  char        s[5];       1     5     2
  long        l;          8     8     0
  struct s2   *next;     16     8     8
  long double ld;        32    16     0
  char        c2;        48     1    15
}
$                                     x86_64
```

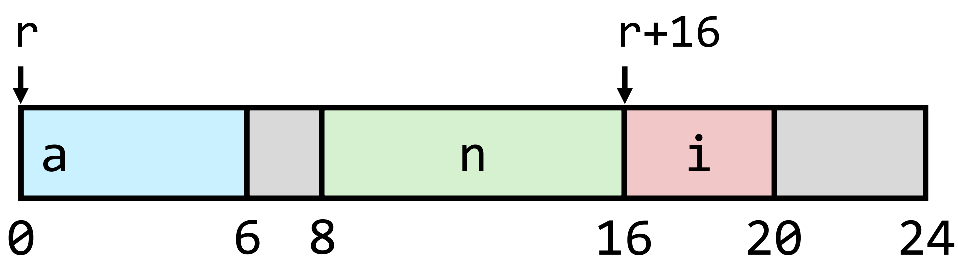
```
$ gcc -m32 -o struct2.32 struct2.c
$ ./struct2.32
                        OFS  SIZE   PAD
struct s2 {                    32
  char        c1;         0     1     0
  char        s[5];       1     5     2
  long        l;          8     4     0
  struct s2   *next;     12     4     0
  long double ld;        16    12     0
  char        c2;        28     1     3
}
$                                       IA32
```

Structure Access

```
struct rec {
  short      a[3];
  struct rec *n;
  int          i;
};
```

- Accessing structure member
  - Pointer r points to first byte of structure
  - Access elements with offsets

```
void set_i(struct rec *r, int val)
{
  r->i = val;
}
```
struct3.c



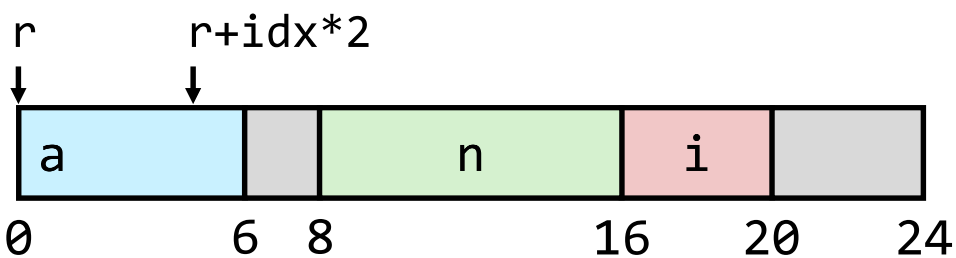

```
set_i:
  sw    a1,16(a0)
  ret
```
struct3.s

Pointer to member

```
struct rec {
  short     a[3];
  struct rec *n;
  int         i;
};
```

- Generating a pointer to a member
  - Pointer r points to first byte of structure
  - Compute pointer using offset to member

```
short *get_ap(struct rec *r, int idx)
{
  return &r->a[idx];
}
```
struct3.c




```
get_ap:
  slli  a1,a1,1
  add   a0,a0,a1
  ret
```
struct3.s

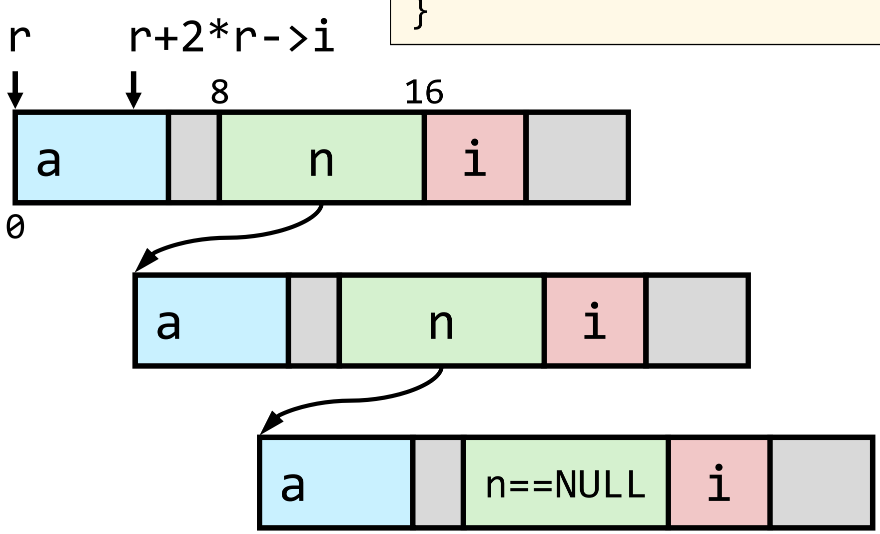
Following a Linked List

```
struct rec {
  short    a[3];
  struct rec *n;
  int         i;
};
```

- Set `r->a[r->i]` to `val` for all elements of the list
  - Pointer `r` points to first byte of structure
  - Compute pointer using offset to member

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

struct3.c



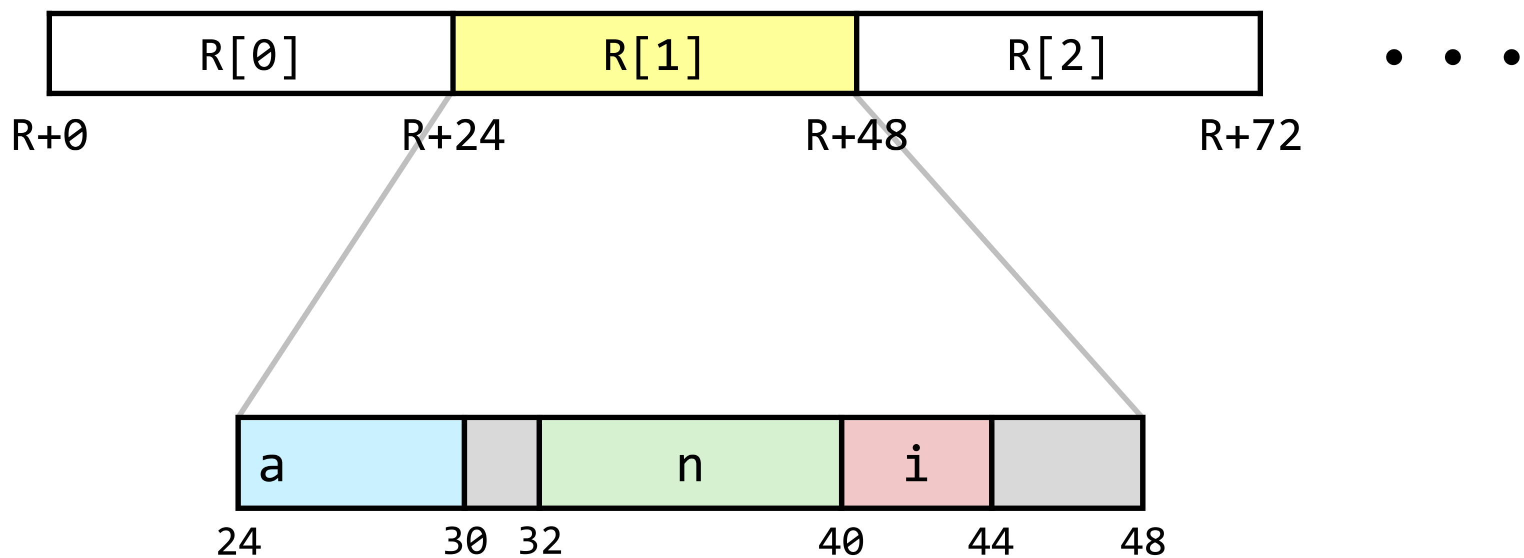

```
get_ap:
  beq   a0,zero,.L1 # r == NULL ?
.L6:
  lw    a5,16(a0)    # a5 = r->i
  slli  a5,a5,1      # a5 = a5*2
  add   a5,a0,a5     # &r->a[i]
  ld    a0,8(a0)     # a0 = r->n
  sh    a1,0(a5)     # r-a[i] = val
  bne   a0,zero,.L3  # r == NULL ?
.L12:
  ret
```

struct3.s

Arrays of Structures

```
struct rec {
  short     a[3];
  struct rec *n;
  int          i;
} R[10];
```

- Structs include padding at end up to next multiple of the struct's alignment requirement
- Allows regular packing of structs into an array

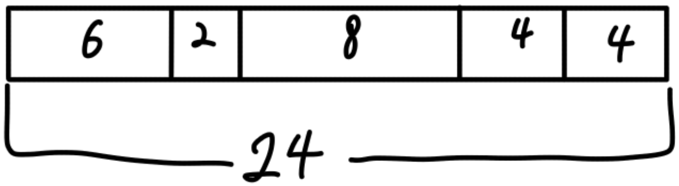


Accessing Members in Struct Arrays

- Compute offset to struct `R[idx]` at runtime
- Add offset to member `i`
  - offset determined statically at compile time

```
struct rec {
  short      a[3]; 6
  struct rec *n;   8
  int        i;    4
} R[10];
```

```
int get_i(int idx)
{
  return R[idx].i;
}
```
struct3.c




```
get_i:
    slli  a5,a0,1        # a5 = idx*2
    add   a5,a5,a0       # a5 = idx*3
&R  lui   a0,%hi(R)
    slli  a5,a5,3        # a5 = idx*24
    addi  a0,a0,%lo(R)   # a0 = &R
    add   a0,a0,a5       # a0 = &R[idx]
    lw    a0,16(a0)      # R[idx]->i
    ret
```
struct3.s

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Avoiding Unnecessary Padding

- Reorder members, putting those with the strongest alignment requirements first

```
struct s2 {
  char         c1;
  char         s[5];
  long         l;
  struct s2    *next;
  long double  ld;
  char         c2;
};
```

struct2.c

RV64

	size	align
struct s2	64	16
c1	1	1
s[5]	5	1
l	8	8
*next	8	8
ld	16	16
c2	1	1

long to small

```
struct s2a {
  long double ld;
  struct s2   *next;
  long        l;
  char        s[5];
  char        c1;
  char        c2;
};
```

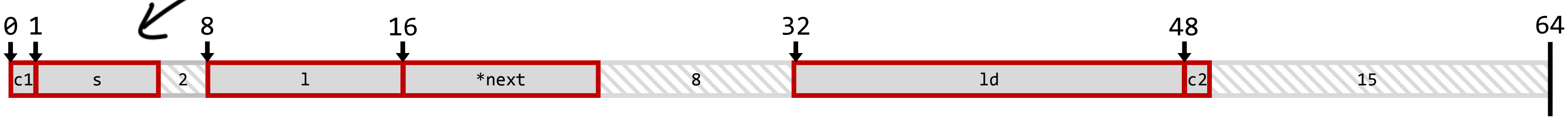
struct2a.c

RV64

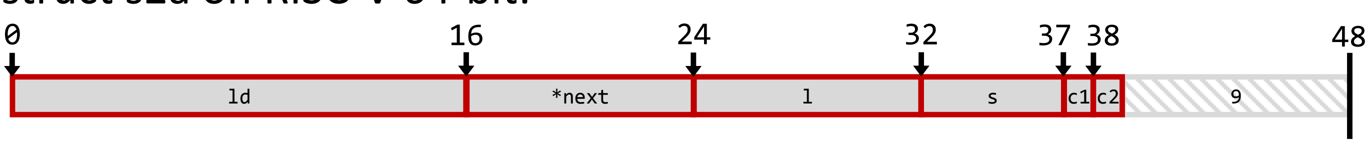
	size	align
struct s2a	48	16
ld	16	16
*next	8	8
l	8	8
s[5]	5	1
c1	1	1
c2	1	1

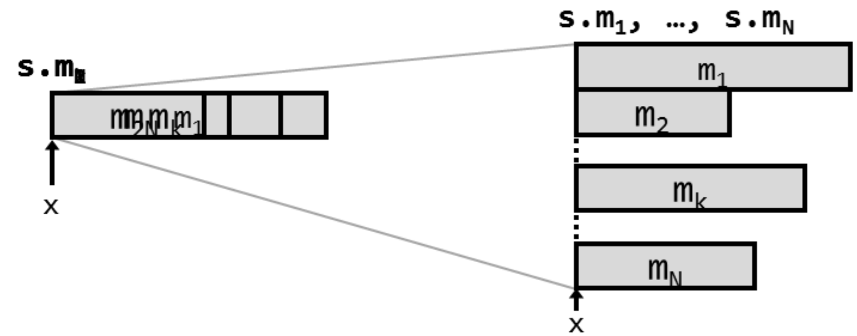
why this thing occur?

struct s2 on RISC-V 64-bit:



struct s2a on RISC-V 64-bit:





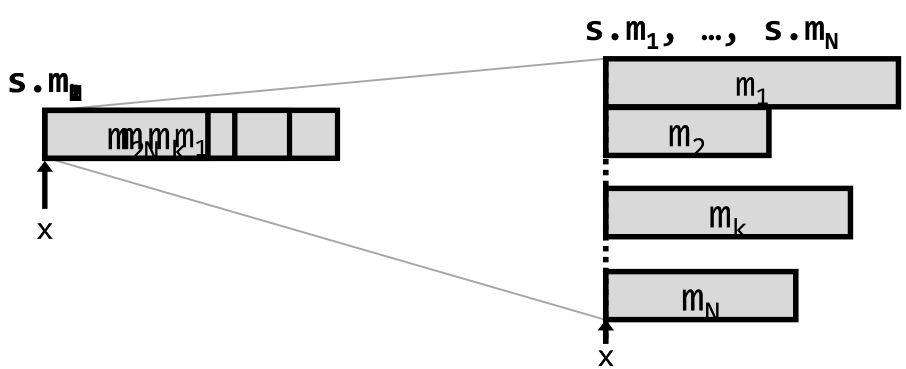
Unions

Unions

- **Declaration**

```
union name {
  <T1> <m1>;
  <T2> <m2>;
  …
  <TN> <mN>;
};
```

- **Memory layout**
  - consecutive memory region containing all members m_i, and properly aligned
  - all members *are located at offset 0 and overlap in memory*



Unions

- **Alignment**
  - union alignment = maximum alignment requirement of any of its members
  - member alignment = alignment requirement of member type

- **Address of k-th member**
  - start of union

$$adr_{m_k} = x$$

- **Size of union**

$$s_u = \max_{1 \le i \le N}(s_{m_i})$$

Example: Using Unions to Access Bit Patterns

- Task: print bit pattern of a floating point number
- Try 1:

```
#include <stdio.h>

unsigned int get_bitpattern(float f)
{
  return (unsigned int)f;
}

void main(void)
{
  float f = 3.14159265358979323846;
  unsigned int u, i;

  u = get_bitpattern(f);

  printf("%12.10f = ", f);
  for (i=sizeof(u)*8; i>0; i--) {
    printf("%c", (u & (1 << (i-1)) ? '1' : '0'));
  }
  printf("b = 0x%08x\n", u);
}
```

union1.c

bit masking

Output looks suspiciously wrong:

```
$ riscv64-unknown-elf-gcc -mabi=lp64d -march=rv64g -O2 -o union1 union1.c
$ spike pk union1
bbl loader
3.1415927410 = 00000000000000000000000000000011b = 0x00000003
```

(yes, unsigned)

Example: Using Unions to Access Bit Patterns

- **Task: print bit pattern of a floating point number**
- Try 1:

```
#include <stdio.h>

unsigned int get_bitpattern(float f)
{
  return (unsigned int)f;
}

void main(void)
{
  float f = 3.14159265358979323846;
  unsigned int u, i;

  u = get_bitpattern(f);

  printf("%12.10f = ", f);
  for (i=sizeof(u)*8; i>0; i--) {
    printf("%c", (u & (1 << (i-1)) ? '1' : '0'));
  }
  printf("b = 0x%08x\n", u);
}
```

union1.c

The assembly reveals

```
get_bitpattern:
  fcvt.wu.s a0,fa0,rtz
  sext.w    a0,a0
  ret
```

fcvt.wu.s: conversion float → 32-bit int
3.1415... → 3

Output looks suspiciously wrong:

```
$ riscv64-unknown-elf-gcc -mabi=lp64d -march=rv64g -O2 -o union1 union1.c
$ spike pk union1
bbl loader
3.1415927410 = 00000000000000000000000000000011b = 0x00000003
```

Example: Using Unions to Access Bit Patterns

f & u share same memory address
But why? (generic casting

- Try 2:

32bit

fu

float

```
#include <stdio.h>

unsigned int get_bitpattern(float f)
{
  union {
    float f;
    unsigned int u;
  } fu;
  fu.f = f;
  return fu.u;
}

void main(void)
{
  float f = 3.14159265358979323846;
  unsigned int u, i;

  u = get_bitpattern(f);

  printf("%12.10f = ", f);
  for (i=sizeof(u)*8; i>0; i--) {
    printf("%c", (u & (1 << (i-1)) ? '1' : '0'));
  }
  printf("b = 0x%08x\n", u);
}
                                                    union2.c
```

This is what we indented:

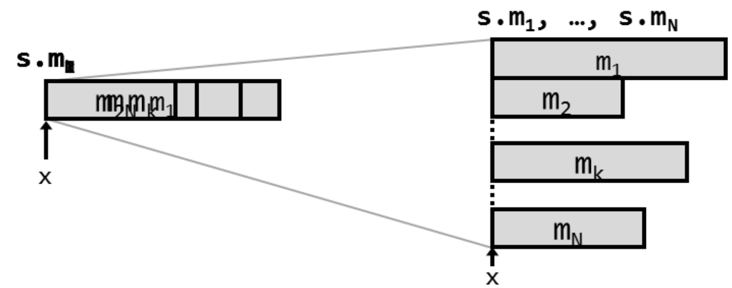
```
get_bitpattern:
  fmv.x.s    a0,fa0
  sext.w     a0,a0
  ret
```

fmv.x.s: move float into int register
(bit pattern preserved)

interesting!

Output:

```
$ riscv64-unknown-elf-gcc -mabi=lp64d -march=rv64g -O2 -o union2 union2.c
$ spike pk union2
bbl loader
3.1415927410 = 01000000010010010000111111011011b = 0x40490fdb
```



Module Summary

Module Summary

- **Data alignment**
  - Data is aligned at certain addresses for performance reasons
  - Basic alignment rule: alignment = size of basic data type

- **Composite data structures**
  - There are no composite data structure at the ISA level!
  - Mapped by the compiler to contiguous allocation of memory
    - Aligned to satisfy every element's alignment requirement
    - Accessed using offsets
    - Variable name = pointer to start of composite data structure

CSE 컴퓨터공학부
Department of Computer Science & Engineering

Module Summary

- **Arrays**
  - No bounds checking
  - Multi-dimensional and multi-level arrays are fundamentally different
    - Multi-dimensional: single contiguous sequence of bytes, row-major layout
    - Multi-level: elements in outer arrays are pointers to inner-level arrays
- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- **Unions**
  - Overlay declarations
  - Can be used to circumvent type system