000000000 01010100 30011100 00002020 20202E4F 52494720 20207833 3030300A E0001300 00002020 20204C45 41202052 1C3015C0 794C696E 6509E200 13000000 20202020 4C454120 2052312C 206D794C 696E6540 60001600 00004C4F 4F502020 52205230 2C205231 2C202330 21F00010 00000020 20202020 20202054 52415020 78323105 24001400 00002020 20204C44 20204C44 20205232 2C207465 726D8014 00160000 00202020 20202020 20414444 2052322C 2052322C 20523002 00002020 20202020 20204252 7A201354 (F506 12 0015 000 02)2020 20202020 20414444 2052312C 2052312C 00120000 00202020 20202020 20202020 2042146 1 022146 4F456 1 000C00 00005354 4F502020 20204841 4C54D0FF 04001000 2031F90F 00746572 6D202020 202E4649 4C4C2020 20784646 44306900 00010000 00697400 00010000 00746100 Lecture x000F - 10/17 00010000 00324000 00010000 00010000 00627200 00010000 00010000 00666100 00010000 00613200 00010000 00323300 00010000 00332D00 00010000 002D6500 00010000 00010000 00636500 00010000 00653200 00010000 00323200 00010000 00323000 00010000 00300000 002A0000 696E6520 202E5354 52494E47 SA202020 20226974 61627261 68324066 6132332D 65636532 32302200 00000000



ECE 220 - Fall 2024

Dr. Ivan Abraham



ECE 220 - Fall 2024

Dr. Ivan Abraham



Last time



- Last time
 - Streams & buffers



- Last time
 - Streams & buffers
 - File I/O



- Last time
 - Streams & buffers
 - File I/O
 - Formatted I/O



- Last time
 - Streams & buffers
 - File I/O
 - Formatted I/O
 - Examples



Last time ullet

Reminders

- Streams & buffers
- File I/O
- Formatted I/O
- Examples



- Last time
 Reminders
 - Streams & buffers
 This lecture concludes material for MT2
 - File I/O
 - Formatted I/O
 - Examples



- Reminders Last time \bullet
 - Streams & buffers This lecture concludes material for MT2
 - File I/O
 - Formatted I/O
 - Examples

- MT2 is on 10/31, plan ahead



- Last time
 Streams & buffers
 File I/O
 Formatted I/O
 Reminders
 This lecture concludes material for MT2
 MT2 is on 10/31, plan ahead
 - Examples
 Drop-deadline is tomorrow



ECE 220 - Fall 2024

Dr. Ivan Abraham

Se

З



 Write a function to transpose a given TSV file and write the output to transposed.tsv

3



- Write a function to transpose a given TSV file and write the output to transposed.tsv
 - The number of rows and columns will be present as the first line of the input file: records.tsv

З



- Write a function to transpose a given TSV file and write the output to transposed.tsv
 - The number of rows and columns will be present as the first line of the input file: records.tsv
 - TSV stands for Tab-Separated-Values.

3



- Write a function to transpose a given TSV file and write the output to transposed.tsv
 - The number of rows and columns will be present as the first line of the input file: records.tsv
 - TSV stands for Tab-Separated-Values.

3

3

4

- Zariski Newton
- Russel
- Maxwell
- 99 Monday
- 43 Sunday
- 72 Saturday
 - Wednesday 32



- Write a function to transpose a given TSV file and write the output to transposed.tsv
 - The number of rows and columns will be present as the first line of the input file: records.tsv
 - TSV stands for Tab-Separated-Values.

3

3

4

- Zariski
- Newton
- Russel
- Maxwell
- Monday 99
- Sunday 43
- 72 Saturday
- 32 Wednesday





- Write a function to transpose a given TSV file and write the output to transposed.tsv
 - The number of rows and columns will be present as the first line of the input file: records.tsv

3 4 Zariski N 43 99 Monday S

3

Zariski

Newton

4

 TSV stands for Tab-Separated-Values.

3

Russel	72 Saturday
Maxwell	32 Wednesday
Newton 72	Russel Maxwell 32
Sunday	Saturday Wednesday

99 Monday

43 Sunday



ECE 220 - Fall 2024

Dr. Ivan Abraham

4



 How about comma-separated values? Let us transpose a matrix stored on disk and write it back to disk.

4



- How about comma-separated values? Let us transpose a matrix stored on disk and write it back to disk.
- The input matrix is in file mat.csv with the first line specifying the number of rows and columns in the matrix.

4



- How about comma-separated values? Let us transpose a matrix stored on disk and write it back to disk.
- The input matrix is in file mat.csv with the first line specifying the number of rows and columns in the matrix.
- Write output to file t mat.csv.

Dr. Ivan Abraham

4



ECE 220 - Fall 2024 Dr.

Dr. Ivan Abraham



 Often useful to the programmer to combine pieces of information into a single abstract unit



- Often useful to the programmer to combine pieces of information into a single abstract unit
- Example(s)



- Often useful to the programmer to combine pieces of information into a single abstract unit
- Example(s)
 - A student could have a name (char[80]), UIN (unsigned long int), year (unsigned int) and GPA (float)



- Often useful to the programmer to combine pieces of information into a single abstract unit
- Example(s)
 - A student could have a name (char[80]), UIN (unsigned long int), year (unsigned int) and GPA (float)
 - A flight could have an altitude (unsigned int), latitude (float), longitude (float), airspeed (float) and airline code (char[20])



ECE 220 - Fall 2024 Dr.

Dr. Ivan Abraham



 Achieved by letting the programmer create their own data type using the struct keyword.



- Achieved by letting the programmer create their own data type using the struct keyword.
- Examples:



- Achieved by letting the programmer create their own data type using the struct keyword.
- Examples:

```
struct student{
    char name[80];
    unsigned long UIN;
    unsigned int year;
    float GPA;
};
```



- Achieved by letting the programmer create their own data type using the struct keyword.
- Examples:

```
struct student{
    char name[80];
    unsigned long UIN;
    unsigned int year;
    float GPA;
};
```

struct flightType{
 char flightCode[20];
 unsigned int altitude;
 float longitude;
 float latitude;
 float airSpeed;



```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```



struct flightType{
 char flightCode[20];
 unsigned int altitude;
 float longitude;
 float latitude;
 unsigned float airSpeed;
};

 A struct allows the user to define a **new** data type that groups together items of types that are *already* defined.



struct flightType{ char flightCode[20]; unsigned int altitude; float longitude; float latitude; unsigned float airSpeed; };

- compiler

• A struct allows the user to define a **new** data type that groups together items of types that are *already* defined.

• *Defining* a struct tells the



- struct flightType{ char flightCode[20]; unsigned int altitude; float longitude; float latitude; unsigned float airSpeed; };
 - compiler

ECE 220 - Fall 2024

• A struct allows the user to define a **new** data type that groups together items of types that are *already* defined.

Defining a struct tells the



Defining structs

- struct flightType{ char flightCode[20]; unsigned int altitude; float longitude; float latitude; unsigned float airSpeed; };
 - compiler

ECE 220 - Fall 2024

• A struct allows the user to define a **new** data type that groups together items of types that are *already* defined.

Defining a struct tells the

• How big the struct is ...



Defining structs

- struct flightType{ char flightCode[20]; unsigned int altitude; float longitude; float latitude; unsigned float airSpeed; };
 - compiler

ECE 220 - Fall 2024

• A struct allows the user to define a **new** data type that groups together items of types that are *already* defined.

Defining a struct tells the

• How big the struct is ...

 How to lay items out in memory ...



Defining structs

- struct flightType{ char flightCode[20]; unsigned int altitude; float longitude; float latitude; unsigned float airSpeed; };
 - However ... no memory allocated yet!

• A struct allows the user to define a **new** data type that groups together items of types that are *already* defined.

Defining a struct tells the compiler

• How big the struct is ...

 How to lay items out in memory ...



```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```

ECE 220 - Fall 2024 Dr.

Dr. Ivan Abraham

8

```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```

 Memory is only allocated when variables are created using the newly defined type.

```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```

 Memory is only allocated when variables are created using the newly defined type.

struct flightType plane;

```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```

 Memory is only allocated when variables are created using the newly defined type.

```
struct flightType plane;
struct student s1;
```

Dr. Ivan Abraham

```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```

 Memory is only allocated when variables are created using the newly defined type.

struct flightType plane;
struct student s1;

 Elements of a struct are called its *members*. Members can be accused using the "dot" notation.

```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```

 Memory is only allocated when variables are created using the newly defined type.

```
struct flightType plane;
struct student s1;
```

 Elements of a struct are called its *members*. Members can be accused using the "dot" notation.

```
plane.altitude = 1000;
```

```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```

 Memory is only allocated when variables are created using the newly defined type.

```
struct flightType plane;
struct student s1;
```

 Elements of a struct are called its *members*. Members can be accused using the "dot" notation.

```
plane.altitude = 1000;
plane.airspeed = 800.0;
```

```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```

 Memory is only allocated when • stru variables are created using the initial newly defined type.

```
struct flightType plane;
struct student s1;
```

 Elements of a struct are called its *members*. Members can be accused using the "dot" notation.

```
plane.altitude = 1000;
plane.airspeed = 800.0;
```

• struct variables can also be initialized at declaration.



```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```

 Memory is only allocated when variables are created using the newly defined type.

```
struct flightType plane;
struct student s1;
```

 Elements of a struct are called its *members*. Members can be accused using the "dot" notation.

```
plane.altitude = 1000;
plane.airspeed = 800.0;
```

• struct variables can also be initialized at declaration.

> struct student s1 = {"Garfield", 123456, 6, 3.5};



```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```

 Memory is only allocated when variables are created using the newly defined type.

```
struct flightType plane;
struct student s1;
```

• Elements of a struct are called its *members*. Members can be accused using the "dot" notation.

```
plane.altitude = 1000;
plane.airspeed = 800.0;
```

- structs

• struct variables can also be initialized at declaration.

> struct student s1 = {"Garfield", 123456, 6, 3.5};

Also possible to create arrays of



```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```

 Memory is only allocated when variables are created using the newly defined type.

```
struct flightType plane;
struct student s1;
```

• Elements of a struct are called its *members*. Members can be accused using the "dot" notation.

```
plane.altitude = 1000;
plane.airspeed = 800.0;
```

• struct variables can also be initialized at declaration.

```
structs
```

- struct student s1 = {"Garfield", 123456, 6, 3.5};
- Also possible to create arrays of
 - struct student $BL3[2] = \{s1,$



```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```

 Memory is only allocated when variables are created using the newly defined type.

struct flightType plane; struct student s1;

 Elements of a struct are called its *members*. Members can be accused using the "dot" notation.

```
plane.altitude = 1000;
plane.airspeed = 800.0;
```

```
structs
```

• struct variables can also be initialized at declaration.

> struct student s1 = {"Garfield", 123456, 6, 3.5};

Also possible to create arrays of

struct student $BL3[2] = \{s1,$ {"Scooby", 234578164, 2, 4.0}};



```
struct flightType{
    char flightCode[20];
    unsigned int altitude;
    float longitude;
    float latitude;
    unsigned float airSpeed;
};
```

 Memory is only allocated when variables are created using the newly defined type.

```
struct flightType plane;
struct student s1;
```

 Elements of a struct are called its *members*. Members can be accused using the "dot" notation.

```
structs
```

```
plane.altitude = 1000;
plane.airspeed = 800.0;
```

• struct variables can also be initialized at declaration.

> struct student s1 = {"Garfield", 123456, 6, 3.5};

Also possible to create arrays of

struct student $BL3[2] = \{s1,$ {"Scooby", 234578164, 2, 4.0}; printf("Name is %s", BL3[1].name);



ECE 220 - Fall 2024 Dr. 1

Dr. Ivan Abraham



 How many bytes of memory should one *instance* of student take?

```
struct student{
    char name[80];
    unsigned long UIN;
    unsigned int year;
    float GPA;
};
struct student s1 =
{"Garfield", 123456, 6, 3.5}
```



 How many bytes of memory should one *instance* of student take?

```
struct student{
    char name[80];
    unsigned long UIN;
    unsigned int year;
    float GPA;
};
struct student s1 =
{"Garfield", 123456, 6, 3.5}
```

	••••
G	s1.name[0]
а	s1.name[1]
	s1.name[78]
	s1.name[79]
123456	s1.UIN
6	s1.year
3.5	s1.gpa



 How many bytes of memory should one *instance* of student take?

```
struct student{
    char name[80];
    unsigned long UIN;
    unsigned int year;
    float GPA;
};
struct student s1 =
{"Garfield", 123456, 6, 3.5}
```

80 + 8 + 4 + 4

	••••
G	s1.name[0]
а	s1.name[1]
	s1.name[78]
	s1.name[79]
123456	s1.UIN
6	s1.year
3.5	s1.gpa



• What if we change the definition to this one?

```
struct student{
    char name[74];
    unsigned long UIN;
    unsigned int year;
    float GPA;
};
```



• What if we change the definition to this one?

```
struct student{
    char name[74];
    unsigned long UIN;
    unsigned int year;
    float GPA;
};
```

Dr. Ivan Abraham

8074 + 8 + 4 + 4 = ?

- Let us check using sizeof function.
 - What happened?



 What if we change the definition to this one?

```
struct student{
    char name[74];
    unsigned long UIN;
    unsigned int year;
    float GPA;
};
```

Compilers will often perform "padding" to align memory. Use the sizeof operator to get accurate results!

8074 + 8 + 4 + 4 = ?

- Let us check using sizeof function.
- What happened?



https://en.cppreference.com/w/c/language/object#Alignment

Why padding is done?

ECE 220 - Fall 2024





Compilers prefer to *align* memory to make operations *faster*. ullet





- Compilers prefer to *align* memory to make operations *faster*. lacksquare
- Memory typically has an access granularity.





- Compilers prefer to *align* memory to make operations *faster*.
- Memory typically has an access granularity.
- Suppose we have **4 byte** memory access granularity. \bullet



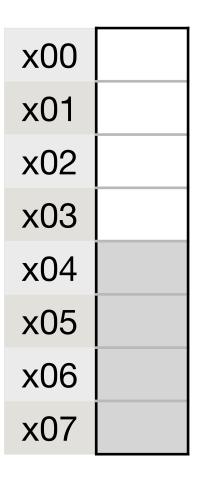


- Compilers prefer to align memory to make operations faster.
- Memory typically has an access granularity.
- Suppose we have 4 byte memory access granularity.
 - Task: Read 4 bytes from address x01





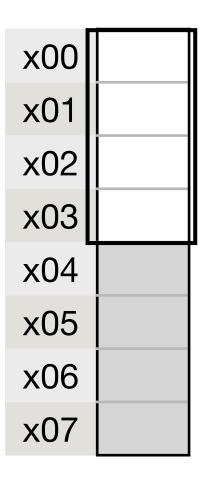
- Compilers prefer to *align* memory to make operations *faster*. ullet
- Memory typically has an access granularity.
- Suppose we have 4 byte memory access granularity.
 - Task: Read 4 bytes from address x01







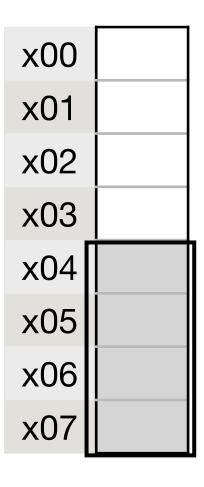
- Compilers prefer to *align* memory to make operations *faster*. •
- Memory typically has an access granularity.
- Suppose we have 4 byte memory access granularity.
 - Task: Read 4 bytes from address x01







- Compilers prefer to align memory to make operations faster. •
- Memory typically has an access granularity.
- Suppose we have 4 byte memory access granularity.
 - Task: Read 4 bytes from address x01

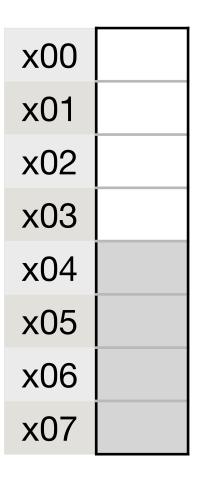


ECE 220 - Fall 2024





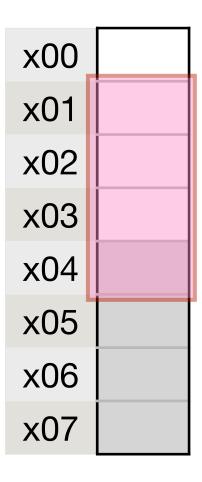
- Compilers prefer to *align* memory to make operations *faster*. ullet
- Memory typically has an access granularity.
- Suppose we have 4 byte memory access granularity.
 - Task: Read 4 bytes from address x01







- Compilers prefer to *align* memory to make operations *faster*. •
- Memory typically has an access granularity. \bullet
- Suppose we have 4 byte memory access granularity.
 - Task: Read 4 bytes from address x01

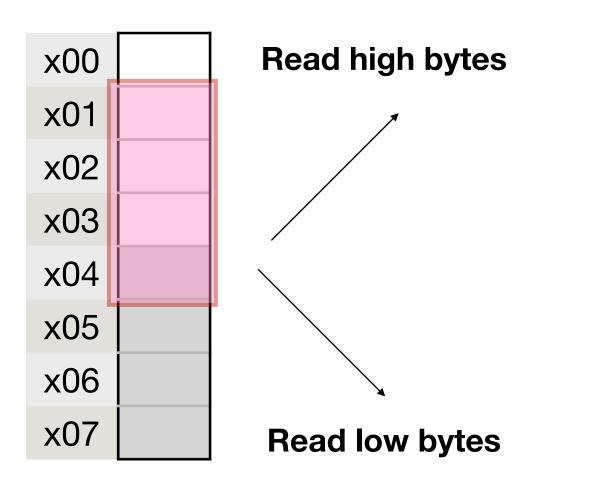


ECE 220 - Fall 2024





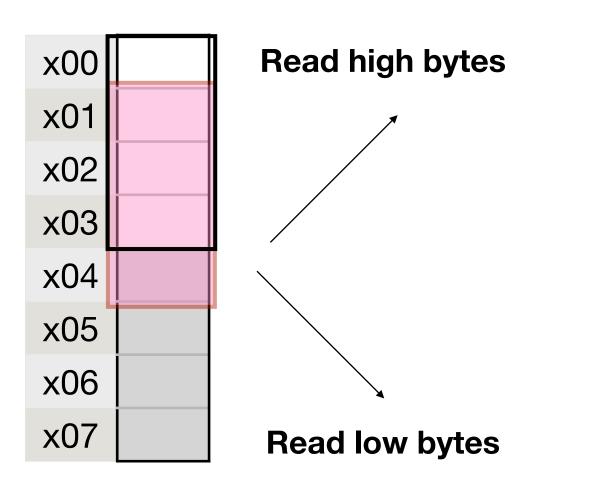
- Compilers prefer to align memory to make operations faster.
- Memory typically has an access granularity.
- Suppose we have 4 byte memory access granularity.
 - Task: Read 4 bytes from address x01







- Compilers prefer to align memory to make operations faster.
- Memory typically has an access granularity.
- Suppose we have 4 byte memory access granularity.
 - Task: Read 4 bytes from address x01



ECE 220 - Fall 2024

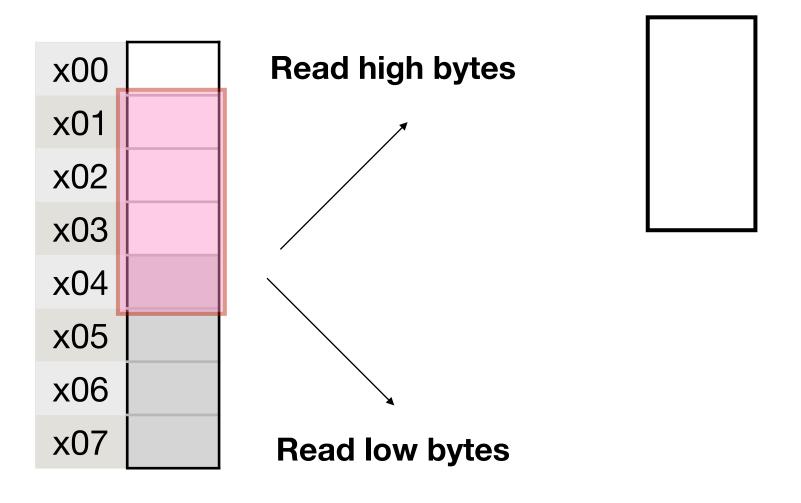
Advanced Topic





11

- Compilers prefer to *align* memory to make operations *faster*. •
- Memory typically has an access granularity.
- Suppose we have 4 byte memory access granularity.
 - Task: Read 4 bytes from address x01



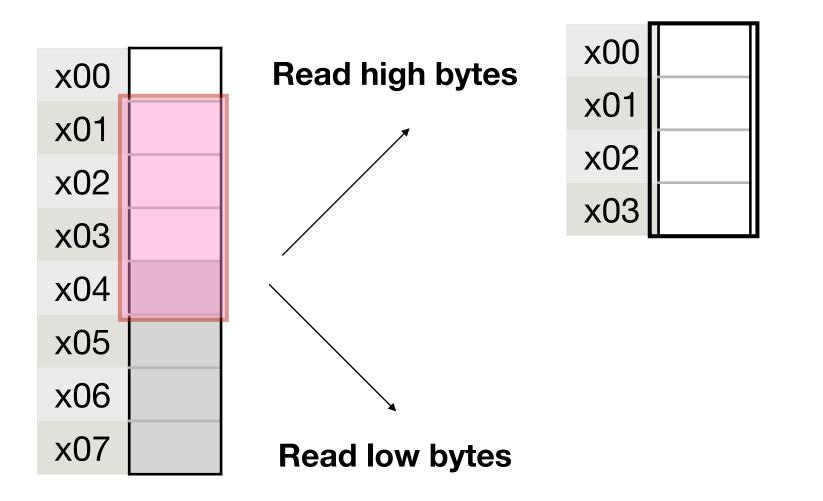
Dr. Ivan Abraham

ECE 220 - Fall 2024





- Compilers prefer to *align* memory to make operations *faster*. ullet
- Memory typically has an access granularity.
- Suppose we have **4 byte** memory access granularity.
 - Task: Read 4 bytes from address x01



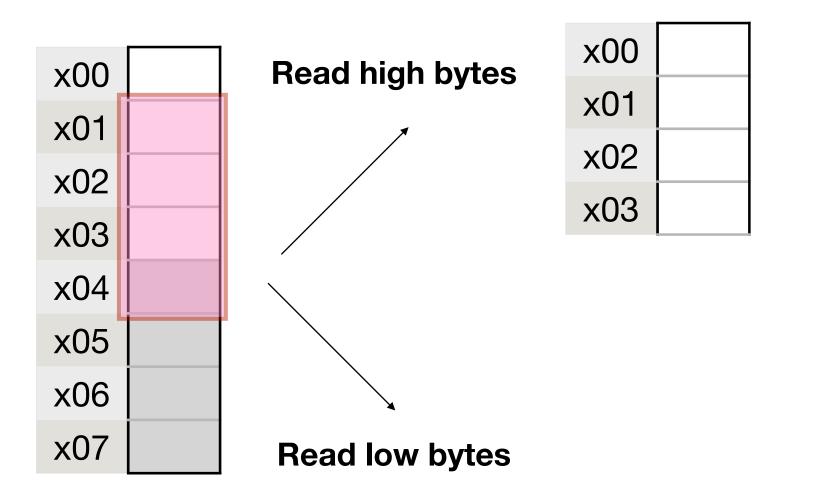
ECE 220 - Fall 2024

Advanced Topic





- Compilers prefer to align memory to make operations faster. lacksquare
- Memory typically has an access granularity.
- Suppose we have **4 byte** memory access granularity.
 - Task: Read 4 bytes from address x01



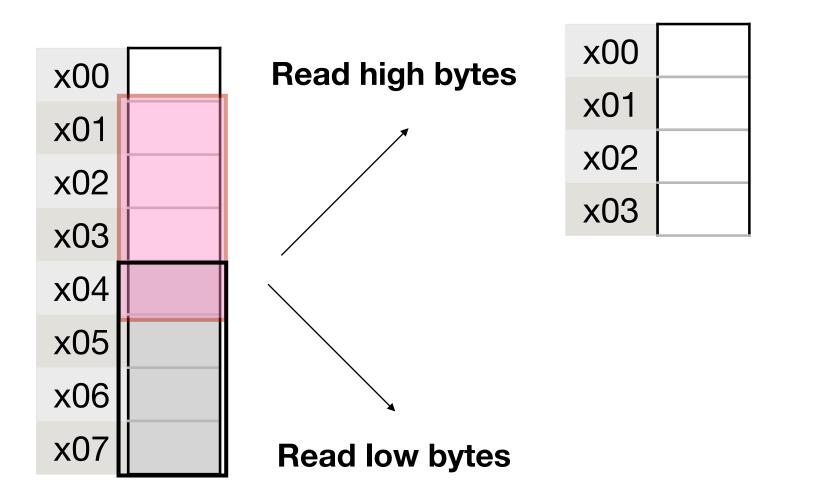
ECE 220 - Fall 2024

Advanced Topic





- Compilers prefer to *align* memory to make operations *faster*. lacksquare
- Memory typically has an access granularity.
- Suppose we have **4 byte** memory access granularity.
 - Task: Read 4 bytes from address x01



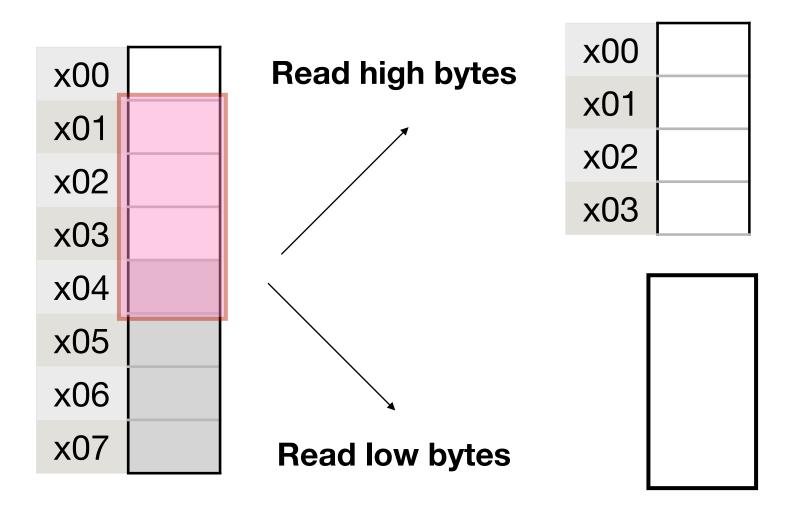
ECE 220 - Fall 2024

Advanced Topic





- Compilers prefer to *align* memory to make operations *faster*. lacksquare
- Memory typically has an access granularity.
- Suppose we have **4 byte** memory access granularity.
 - Task: Read 4 bytes from address x01



ECE 220 - Fall 2024

Dr. Ivan Abraham

11

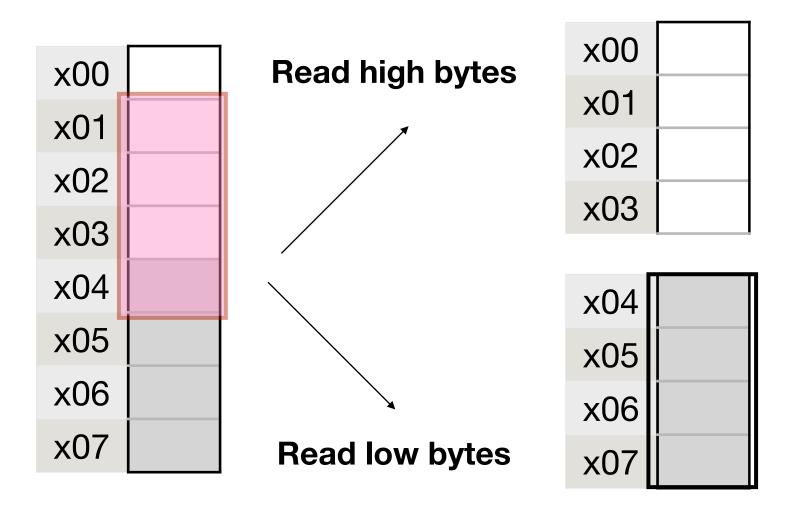
Advanced Topic





11

- Compilers prefer to *align* memory to make operations *faster*. ullet
- Memory typically has an access granularity.
- Suppose we have **4 byte** memory access granularity.
 - Task: Read 4 bytes from address x01



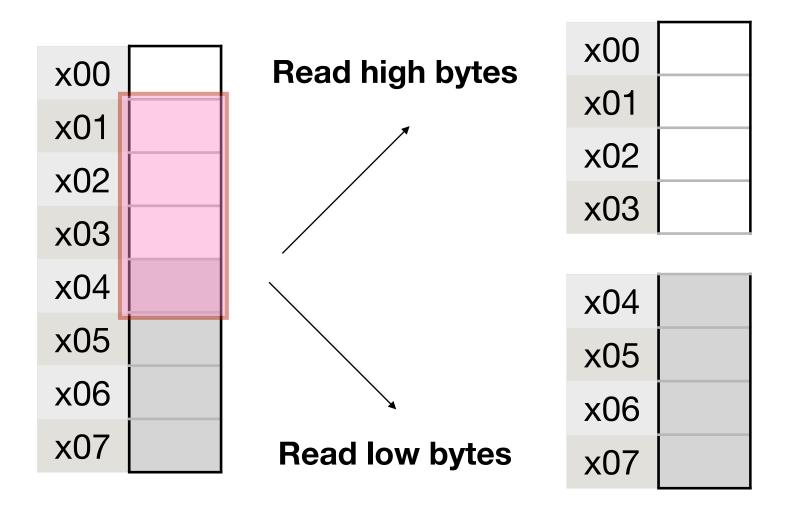
Dr. Ivan Abraham

ECE 220 - Fall 2024





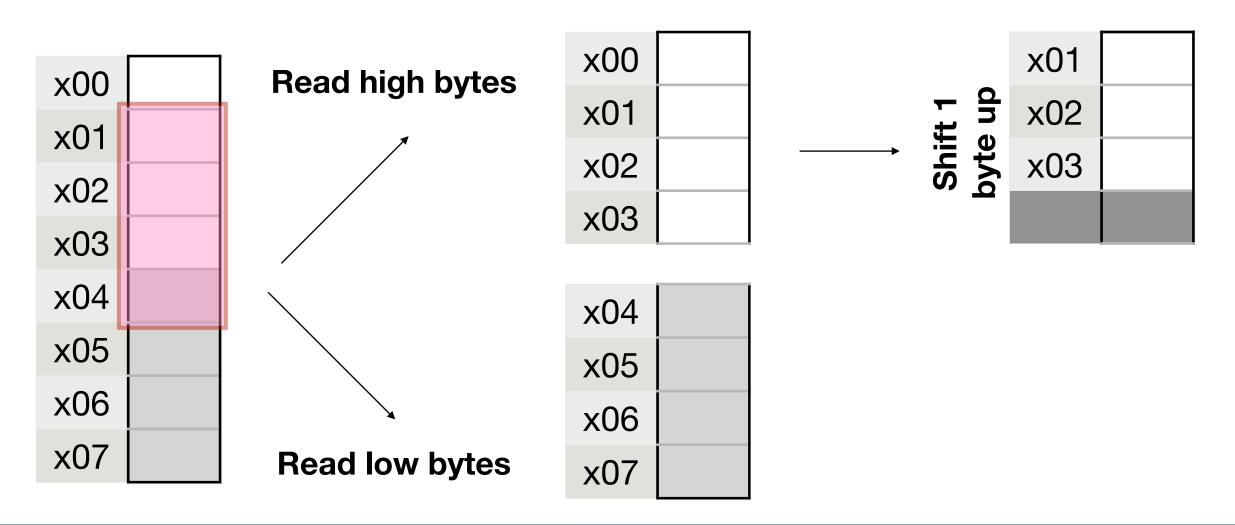
- Compilers prefer to *align* memory to make operations *faster*. ullet
- Memory typically has an access granularity.
- Suppose we have 4 byte memory access granularity.
 - Task: Read 4 bytes from address x01







- Compilers prefer to *align* memory to make operations *faster*. ullet
- Memory typically has an access granularity. \bullet
- Suppose we have 4 byte memory access granularity.
 - Task: Read 4 bytes from address x01

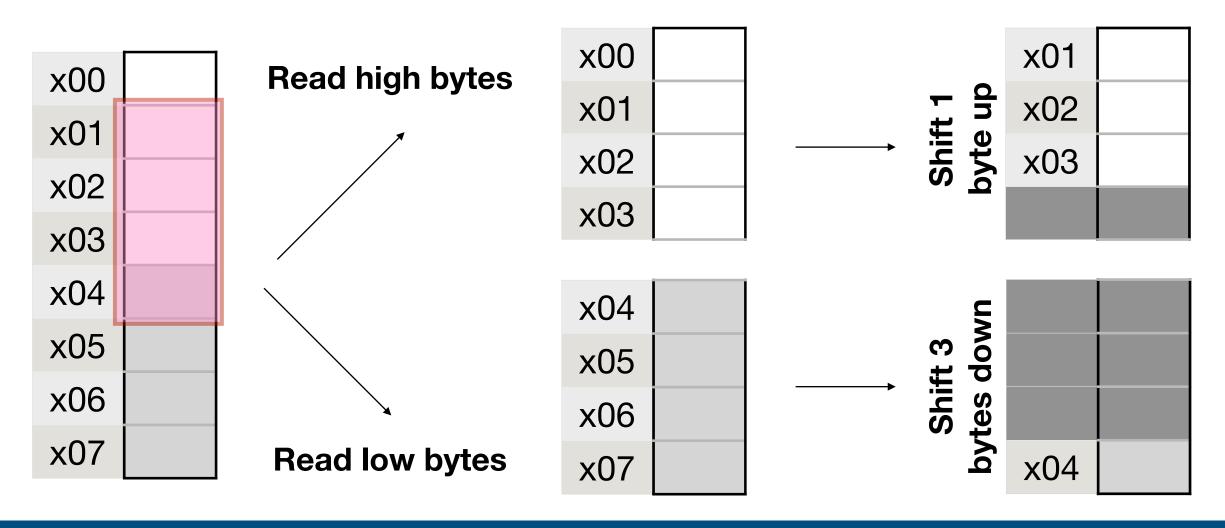


ECE 220 - Fall 2024 Dr. Ivan Abraham 11

Advanced Topic



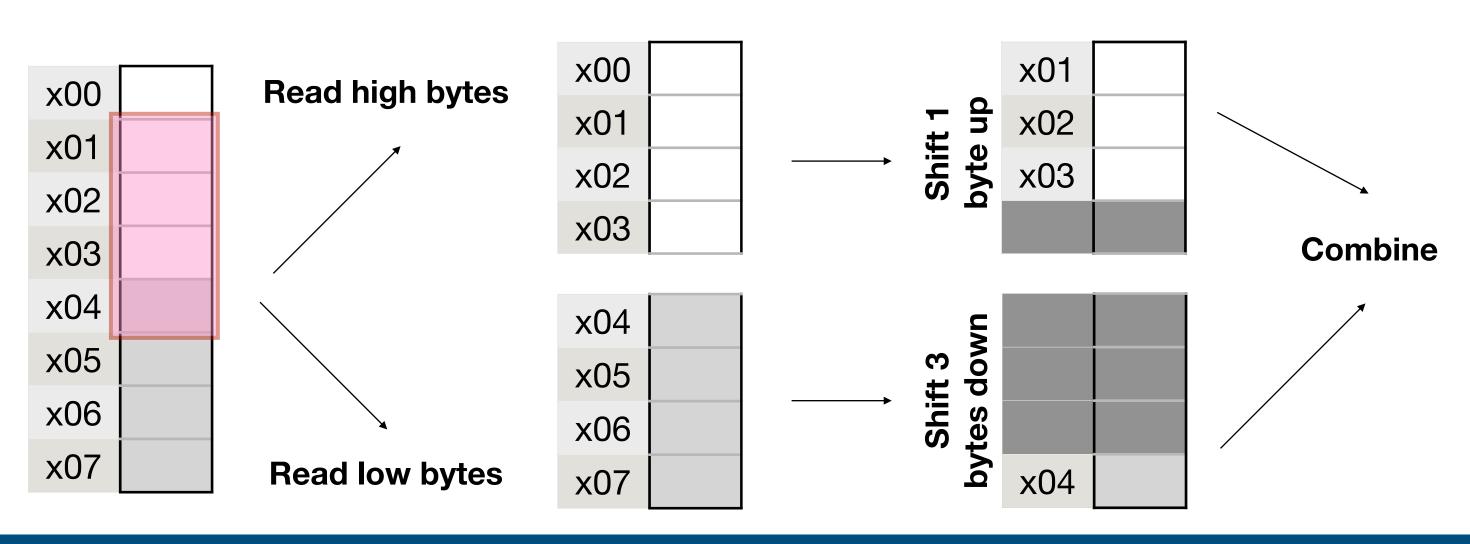
- Compilers prefer to *align* memory to make operations *faster*. ullet
- Memory typically has an access granularity. lacksquare
- Suppose we have **4 byte** memory access granularity. \bullet
 - Task: Read 4 bytes from address x01



ECE 220 - Fall 2024 Dr. Ivan Abraham **Advanced Topic** 11



- Compilers prefer to *align* memory to make operations *faster*. ullet
- Memory typically has an access granularity.
- Suppose we have **4 byte** memory access granularity.
 - Task: Read 4 bytes from address x01

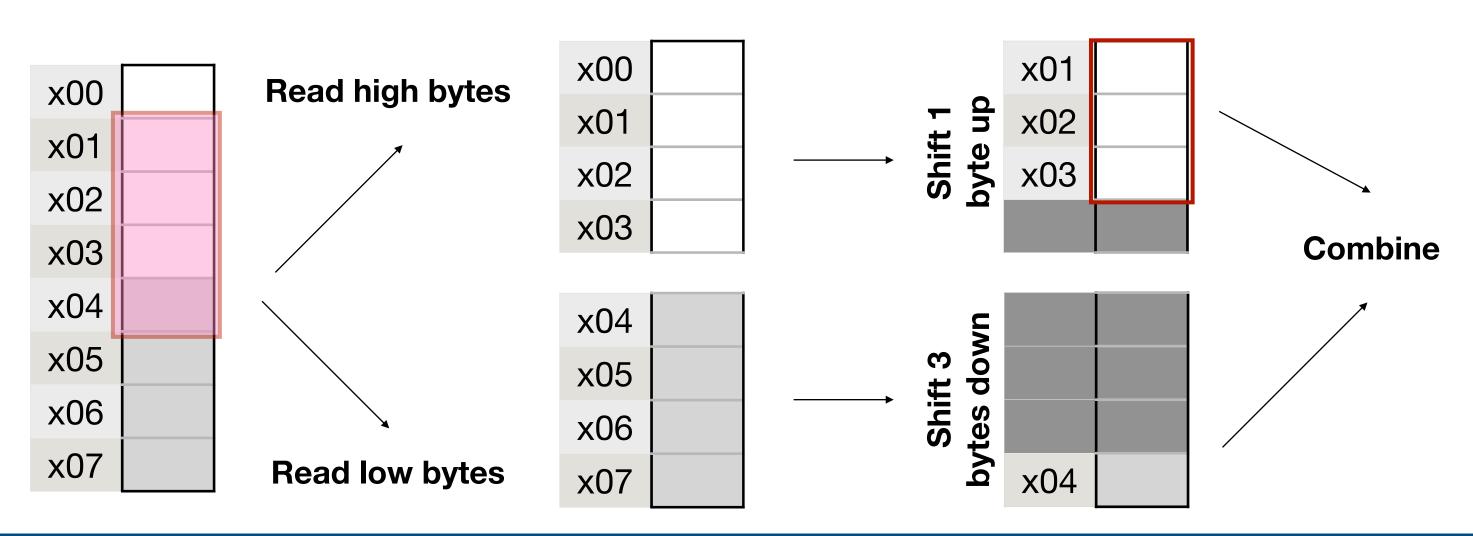


ECE 220 - Fall 2024 **Dr. Ivan Abraham** 11





- Compilers prefer to *align* memory to make operations *faster*. ullet
- Memory typically has an access granularity.
- Suppose we have **4 byte** memory access granularity.
 - Task: Read 4 bytes from address x01

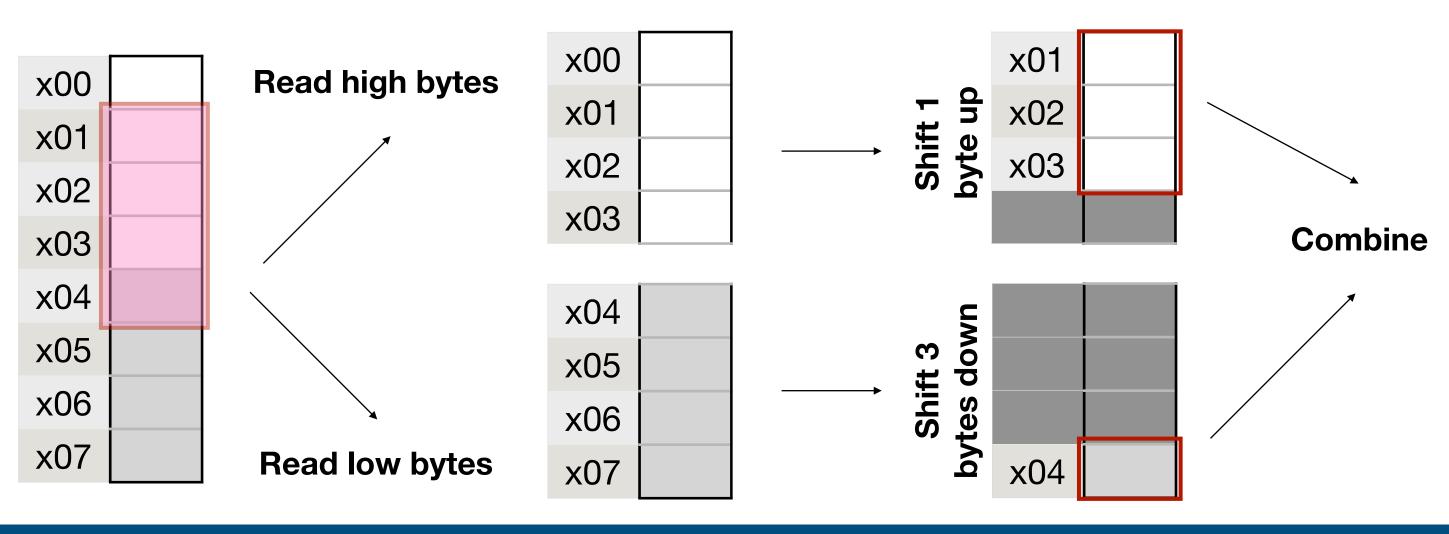


ECE 220 - Fall 2024 Dr. Ivan Abraham 11





- Compilers prefer to *align* memory to make operations *faster*. ullet
- Memory typically has an access granularity.
- Suppose we have **4 byte** memory access granularity.
 - Task: Read 4 bytes from address x01

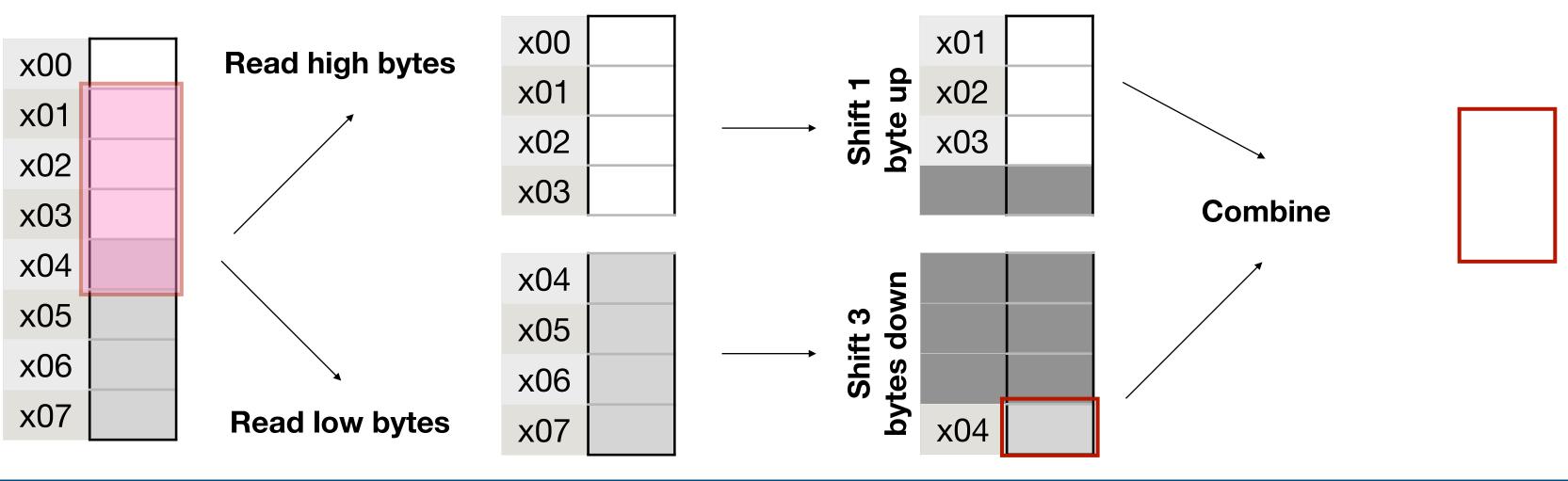


ECE 220 - Fall 2024 Dr. Ivan Abraham 11





- Compilers prefer to *align* memory to make operations *faster*. ullet
- Memory typically has an access granularity.
- Suppose we have **4 byte** memory access granularity.
 - Task: Read 4 bytes from address x01

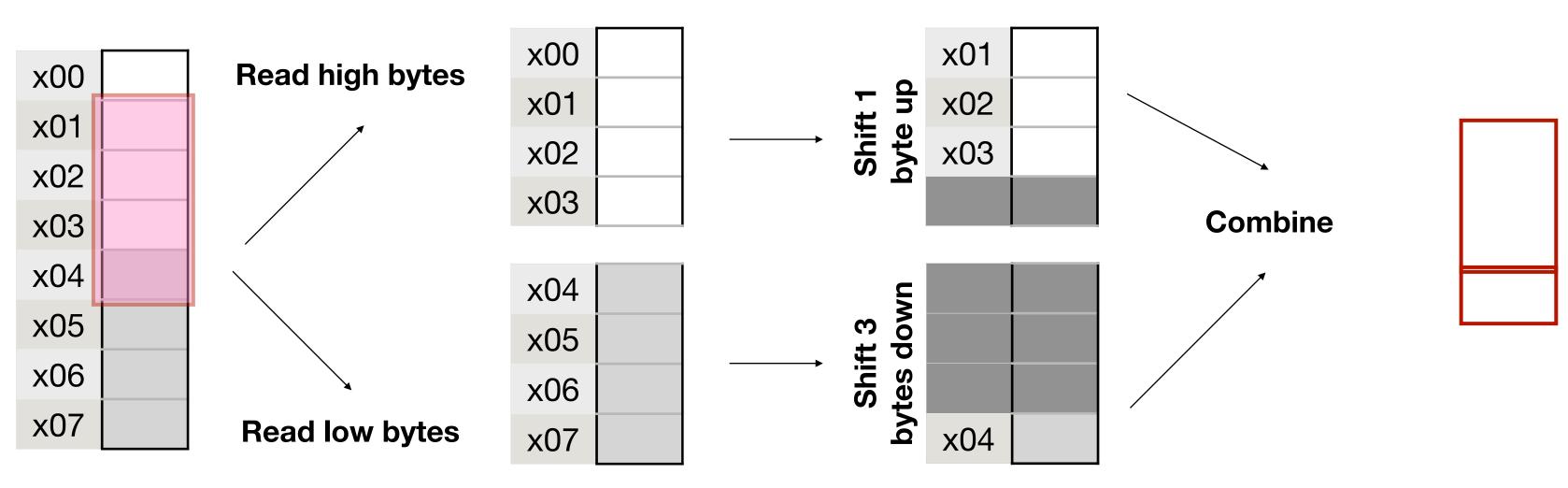


ECE 220 - Fall 2024 Dr. Ivan Abraham 11





- Compilers prefer to *align* memory to make operations *faster*. ullet
- Memory typically has an access granularity.
- Suppose we have **4 byte** memory access granularity.
 - Task: Read 4 bytes from address x01

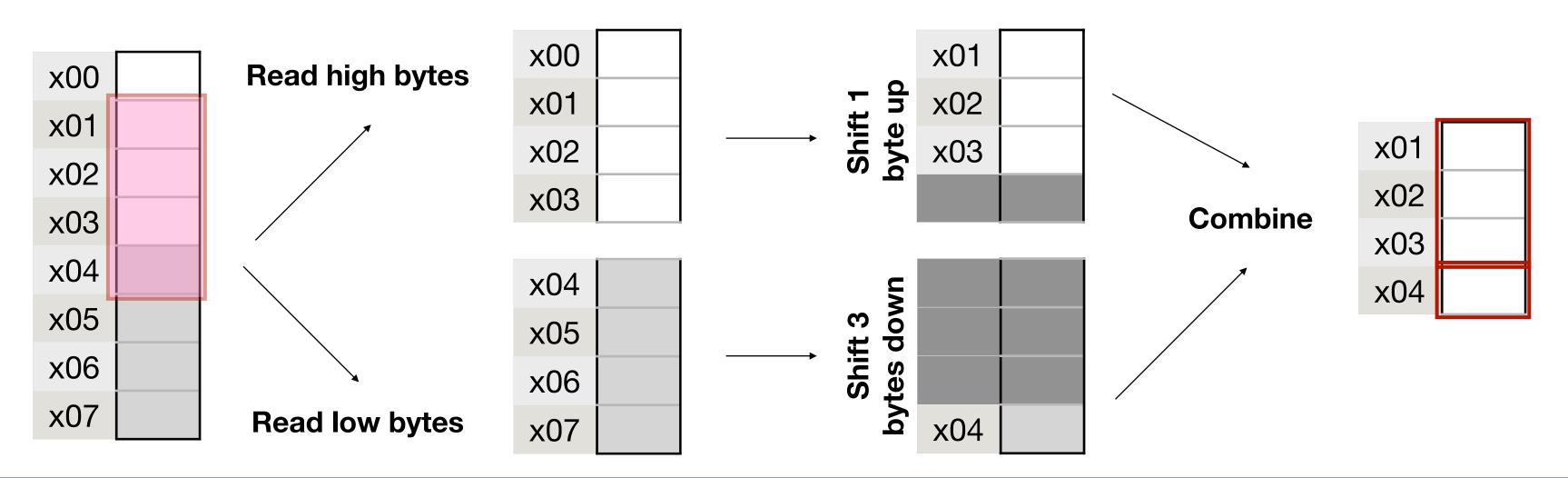


ECE 220 - Fall 2024 Dr. Ivan Abraham 11





- Compilers prefer to *align* memory to make operations *faster*. •
- Memory typically has an access granularity. lacksquare
- Suppose we have **4 byte** memory access granularity.
 - Task: Read 4 bytes from address x01



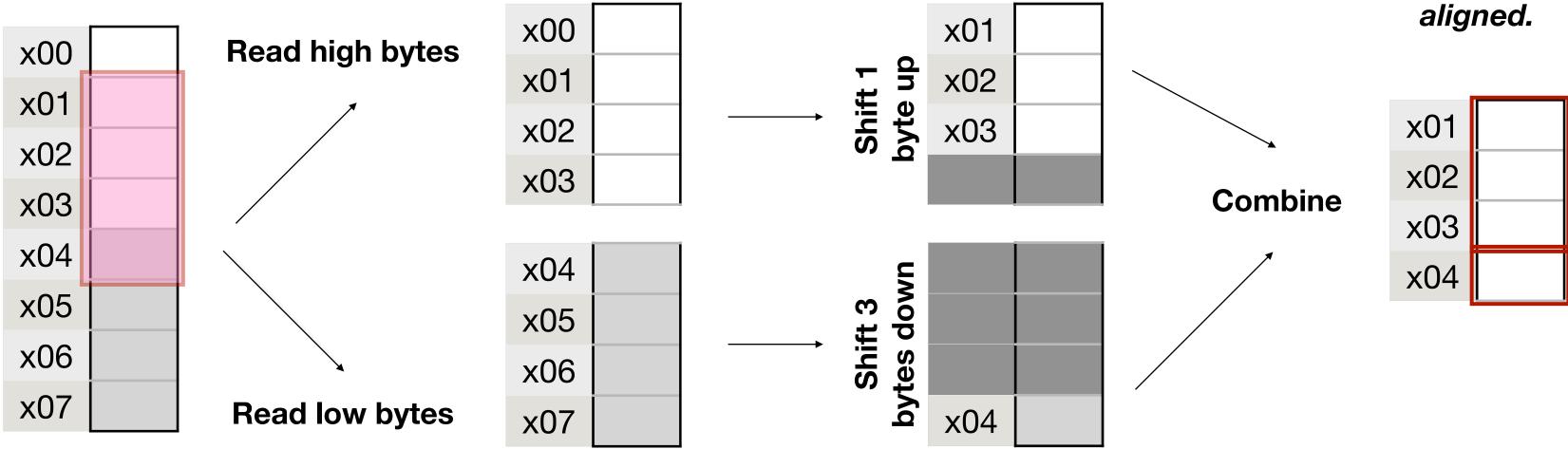
ECE 220 - Fall 2024 **Dr. Ivan Abraham** 11

https://en.cppreference.com/w/c/language/object#Alignment

ILLINOIS



- Compilers prefer to *align* memory to make operations *faster*. ullet
- Memory typically has an access granularity. lacksquare
- Suppose we have **4 byte** memory access granularity.
 - Task: Read 4 bytes from address x01



ECE 220 - Fall 2024 Dr. Ivan Abraham 11

https://en.cppreference.com/w/c/language/object#Alignment

Thus, operations will be faster if memory is

ILLINOIS



ECE 220 - Fall 2024

Dr. Ivan Abraham



• Note how we declared a struct variable:



• Note how we declared a struct variable:

struct flightType plane;



• Note how we declared a struct variable:

struct flightType plane;
struct student s1;



• Note how we declared a struct variable:

struct flightType plane;
struct student s1;

 Annoying to keep having to say struct xyz, struct abc more so in the context of function calls



• Note how we declared a struct variable:

```
struct flightType plane;
struct student s1;
```

- Annoying to keep having to say struct xyz, struct abc more so in the context of function calls
- C provides a mechanism to avoid this verbosity.



- Note how we declared a struct variable:
 struct flightType plane; struct student s1;
 Annoying to keep having to say struct xyz, struct abc more so in the context of function calls
 typedef s typedef s char unsig float float struct flightType plane; struct student s1;
- C provides a mechanism to avoid this verbosity.

ECE 220 - Fall 2024

typedef struct flightType{
 char flightCode[20];
 unsigned int altitude;
 float longitude;
 float latitude;
 unsigned float airSpeed;
} Flight;



ECE 220 - Fall 2024

Dr. Ivan Abraham



• One can define pointers to structs the usual way.

```
Flight planes[100];
Flight *ptr1;
ptr1 = &planes[10];
Flight *ptr2;
ptr2 = planes;
```

Dr. Ivan Abraham



- One can define pointers to structs the usual way.
- pointers you can

```
Flight planes[100];
Flight *ptr1;
ptr1 = &planes[10];
Flight *ptr2;
ptr2 = planes;
```

Dr. Ivan Abraham

To access struct elements via



- One can define pointers to structs the usual way.
- pointers you can

```
    Dereference and dot
```

```
Flight planes[100];
Flight *ptr1;
ptr1 = &planes[10];
Flight *ptr2;
ptr2 = planes;
```

Dr. Ivan Abraham

To access struct elements via



• One can define pointers to structs the usual way.

```
pointers you can
```

```
Flight planes[100];
Flight *ptr1;
ptr1 = &planes[10];
Flight *ptr2;
ptr2 = planes;
```

Dereference and dot

Dr. Ivan Abraham

To access struct elements via

printf("I am %f feet high",



• One can define pointers to structs the usual way.

```
pointers you can
```

```
Flight planes[100];
Flight *ptr1;
ptr1 = &planes[10];
Flight *ptr2;
ptr2 = planes;
```

Dereference and dot

```
ECE 220 - Fall 2024
```

Dr. Ivan Abraham

To access struct elements via

printf("I am %f feet high", (*ptr1).altitude);



• One can define pointers to structs the usual way.

```
Flight planes[100];
Flight *ptr1;
ptr1 = &planes[10];
Flight *ptr2;
ptr2 = planes;
```

- pointers you can
 - Dereference and dot

```
    Arrow
```

Dr. Ivan Abraham

To access struct elements via

printf("I am %f feet high", (*ptr1).altitude);



• One can define pointers to structs the usual way.

```
Flight planes[100];
Flight *ptr1;
ptr1 = &planes[10];
Flight *ptr2;
ptr2 = planes;
```

- pointers you can
 - Dereference and dot
 - Arrow

To access struct elements via

printf("I am %f feet high", (*ptr1).altitude);

printf("I am %f feet high",



• One can define pointers to structs the usual way.

```
Flight planes[100];
Flight *ptr1;
ptr1 = &planes[10];
Flight *ptr2;
ptr2 = planes;
```

- pointers you can
 - Dereference and dot
 - Arrow

printf("I am %f feet high", ptr1->altitude);

To access struct elements via

printf("I am %f feet high", (*ptr1).altitude);



- One can define pointers to structs the usual way.
- pointers you can

```
Flight planes[100];
Flight *ptr1;
ptr1 = &planes[10];
Flight *ptr2;
ptr2 = planes;
```

- Dereference and dot

```
    Arrow
```

printf("I am %f feet high", Special syntax! ptr1->altitude);

ECE 220 - Fall 2024

Dr. Ivan Abraham

To access struct elements via

printf("I am %f feet high", (*ptr1).altitude);



One can write function definitions involving using structs in either lacksquareway:



 One can write function definitions involving using structs in either way:

```
void print student(struct student s){
  printf("Student %s is associated with UIN: %lu\n", s.name, s.UIN);
 printf("%s is in Year %d with GPA %f\n", s.name, s.year, s.GPA);
}
```



 One can write function definitions involving using structs in either way:

```
void print student(struct student s){
  printf("Student %s is associated with UIN: %lu\n", s.name, s.UIN);
 printf("%s is in Year %d with GPA %f\n", s.name, s.year, s.GPA);
}
```

```
void print flight(Flight f){
  printf("Flight #%s is at altitude %u\n", f.flightCode, f.altitude);
  printf("%s has speed %f\n", f.flightCode, f.airSpeed);
}
```



• We could also pass the struct via reference:



Passing structs as arguments

• We could also pass the struct via reference:

```
void print flight loc(Flight *f) {
  printf("Flight #%s is at altitude %u\n", f->flightCode, f->altitude);
  printf("%s has lattitude: %f\n", f->flightCode, f->latitude);
  printf("%s has longitude: %f\n", f->flightCode, f->longitude);
}
```

Dr. Ivan Abraham



Passing structs as arguments

• We could also pass the struct via reference:

```
void print flight loc(Flight *f) {
  printf("Flight #%s is at altitude %u\n", f->flightCode, f->altitude);
  printf("%s has lattitude: %f\n", f->flightCode, f->latitude);
  printf("%s has longitude: %f\n", f->flightCode, f->longitude);
}
```

• Which is cheaper in terms of memory/run-time stack?



Passing structs as arguments

• We could also pass the struct via reference:

```
void print flight loc(Flight *f) {
  printf("Flight #%s is at altitude %u\n", f->flightCode, f->altitude);
  printf("%s has lattitude: %f\n", f->flightCode, f->latitude);
  printf("%s has longitude: %f\n", f->flightCode, f->longitude);
}
```

- Which is cheaper in terms of memory/run-time stack?
 - What if we had an array of structs?



Structs within structs

ECE 220 - Fall 2024

Dr. Ivan Abraham



Structs within structs

• Nothing stops us from creating a struct composed of structs.

Suppose we have:

```
struct geoloc{
   float lattitude;
   float longitude;
};
```



Structs within structs

 Nothing stops us from creating 	• Then v
a struct composed of structs.	typed
	cha
Suppose we have:	uns
	uns
<pre>struct geoloc{</pre>	sti
<pre>float lattitude;</pre>	sti
<pre>float longitude;</pre>	} Fli
};	

we can do:

edef struct flight{
ar code[8];
asigned int arrival_time;
asigned int depart_time;
ruct geoloc origin;
ruct geoloc destination;
.ight;



ECE 220 - Fall 2024

Dr. Ivan Abraham



• Writing a struct to a file:



• Writing a struct to a file:

fwrite(void *ptr, size, n_memb, FILE *stream)



• Writing a struct to a file:

fwrite(void *ptr, size, n memb, FILE *stream)

• ptr is pointer to instance of the struct to write



• Writing a struct to a file:

fwrite(void *ptr, size, n memb, FILE *stream)

- ptr is pointer to instance of the struct to write
- size is the size in bytes of each element to be written (use size of)



• Writing a struct to a file:

fwrite(void *ptr, size, n memb, FILE *stream)

- ptr is pointer to instance of the struct to write
- size is the size in bytes of each element to be written (use size of)
- n memb is the number of items to write, each with size of size bytes



• Writing a struct to a file:

fwrite(void *ptr, size, n memb, FILE *stream)

- ptr is pointer to instance of the struct to write
- size is the size in bytes of each element to be written (use size of)
- n memb is the number of items to write, each with size of size bytes
- stream is the pointer to FILE object in *binary write mode*.



• Writing a struct to a file:

fread(void *ptr, size, n memb, FILE *stream)

- ptr is pointer to instance of the struct to hold data
- size is the size in bytes of each element to be read (use size of)
- n memb is the number of items to read, each with size of size bytes
- stream is the pointer to FILE object in binary read mode.



Exercise

- In a C file, use a loop and have the user input three records of the Flight struct.
 - Write this data to disk using fwrite.
- In another C file, read the data back to an array of Flight using fread.

```
struct geoloc{
   float lattitude;
   float longitude;
};
```

```
typedef struct flight{
   char code[8];
   unsigned int arrival_time;
   unsigned int depart_time;
   struct geoloc origin;
   struct geoloc destination;
} Flight;
```

