

Automatic Transformation of Bit-Level C Code to Support Multiple Equivalent Data Layouts

Marius Nita and Dan Grossman

University of Washington WASP Group

wasp.cs.washington.edu

Good Things About C

- Suitable for writing bit-level code:
 - Debuggers
 - Runtime systems
 - Operating system kernels
 - Embedded systems
- **Portable**: compilers for virtually every platform.

But...

- Not particularly well-suited for writing *portable* bit-level code.
- Must support multiple layouts of the same data, e.g.:
 - Bit-, byte- and bit field-level endianness.
- A function of
 - Hardware
 - Binary data formats

Standard Approach: Duplication

- A copy of the code for each layout.
- An **if** or **#ifdef** chooses between the copies.

```
if (big_endian) {  
    big endian version  
} else {  
    little endian version  
}
```

Problems With Duplication

- Writing and maintaining code copies is error-prone:
 - Code is inherently low-level.
 - Layout differences are ***subtle*** and ***implicit***.
 - Copies must be changed in sync.

A Real Example From GDB

```
if (endianness(abfd) == BIG) {
    type = ( (extern ? 0x10 : 0)
            | (pcrel  ? 0x80 : 0)
            | (neg    ? 0x08 : 0)
            | (length << 5));
} else {
    type = ( (extern ? 0x08 : 0)
            | (pcrel  ? 0x01 : 0)
            | (neg    ? 0x10 : 0)
            | (length << 1));
}
```

A Real Example From GDB

```
if (endianness(abfd) == BIG) {
    type = ( (extern ? 0x10 : 0)
            | (pcrel ? 0x80 : 0)
            | (neg ? 0x08 : 0)
            | (length << 5));
} else {
    type = ( (extern ? 0x08 : 0)
            | (pcrel ? 0x01 : 0)
            | (neg ? 0x10 : 0)
            | (length << 1));
}
```

Big



A Real Example From GDB

```
if (endianness(abfd) == BIG) {
    type = ( (extern ? 0x10 : 0)
            | (pcrel ? 0x80 : 0)
            | (neg ? 0x08 : 0)
            | (length << 5));
} else {
    type = ( (extern ? 0x08 : 0)
            | (pcrel ? 0x01 : 0)
            | (neg ? 0x10 : 0)
            | (length << 1));
}
```

Big



Little



Programmers Duplicate

- Widespread evidence of duplication.
 - E.g., CodeSearch for “#ifdef BIG”
- Our GDB/BFD case study:
 - Found thousands of lines of duplicated code.

Programmers Duplicate

- Widespread evidence of duplication.
 - E.g., CodeSearch for “`#ifdef BIG`”
- Our GDB/BFD case study:
 - Found thousands of lines of duplicated code.
- Evidence of bug reports (including GDB/BFD).
 - E.g., google “endianness bug”.
 - Often due to inconsistent updates.

Why Do They Not Abstract?

Why Do They Not Abstract?

```
#define PUT_BITS(endn, size, data, len, shift) \  
    ((endn) == BIG ? \  
        ((data) & ((1 << (len)) - 1)) << (shift) \  
    : ((data) & ((1 << (len)) - 1)) \  
        << (size) - (len) - (shift))
```

...

```
end = endianness(abfd);  
size = sizeof(type);  
type = PUT_BITS(end, size, extern, 1, 4)  
      | PUT_BITS(end, size, pcrel, 1, 7)  
      | PUT_BITS(end, size, neg, 1, 3)  
      | PUT_BITS(end, size, length, 2, 5);
```

Why Do They Not Abstract?

```
#define PUT_BITS(endn,size,data,len,shift) \
    ((endn) == BIG ? \
        ((data)&((1<<(len))-1)) << (shift) \
    : ((data)&((1<<(len))-1)) \
        << (size)-(len)-(shift))
```

...

```
end = endianness(abfd);
size = sizeof(type);
type = PUT_BITS(end, size, extern, 1, 4)
      | PUT_BITS(end, size, pcrel, 1, 7)
      | PUT_BITS(end, size, neg, 1, 3)
      | PUT_BITS(end, size, length, 2, 5);
```

Why Do They Not Abstract?

```
#define PUT_BITS(endn, size, data, len, shift) \  
    ((endn) == BIG ? \  
        ((data) & ((1 << (len)) - 1)) << (shift) \  
    : ((data) & ((1 << (len)) - 1)) \  
        << (size) - (len) - (shift))
```

...

```
end = endianness(abfd);
```

```
size = sizeof(type);
```

```
type = PUT_BITS(end, size, extern, 1, 4)  
      | PUT_BITS(end, size, pcrel, 1, 7)  
      | PUT_BITS(end, size, neg, 1, 3)  
      | PUT_BITS(end, size, length, 2, 5);
```

Why Do They Not Abstract?

- C is **good** at expressing bit-level algorithms clearly and concisely.
 - Abstraction hides the tricky details.

Why Do They Not Abstract?

- C is **good** at expressing bit-level algorithms clearly and concisely.
 - Abstraction hides the tricky details.
- More prone to breaking code that is known to work.

Why Do They Not Abstract?

- C is **good** at expressing bit-level algorithms clearly and concisely.
 - Abstraction hides the tricky details.
- More prone to breaking code that is known to work.
- Hard to define a general set of useful abstractions.

Why Do They Not Abstract?

```
type = ( (extern ? 0x10 : 0)
         | (pcrel  ? 0x80 : 0)
         | (neg    ? 0x08 : 0)
         | (length << 5));
```

Why Do They Not Abstract?

```
type = ( (extern ? 0x10 : 0)
         | (pcrel  ? 0x80 : 0)
         | (neg    ? 0x08 : 0)
         | (length << 5));
```

```
type = ( (extern ? 0x10 : 0)
         | (pcrel  ? 0x80 : 0)
         | (neg    ? 0x08 : 0)
         | (length << 5));
```

Why Do They Not Abstract?

```
type = ( (extern ? 0x10 : 0)
         | (pcrel  ? 0x80 : 0)
         | (neg    ? 0x08 : 0)
         | (length << 5));
```

```
type = ( (extern ? 0x10 : 0)
         | (pcrel  ? 0x80 : 0)
         | (neg    ? 0x08 : 0)
         | (length << 5));
```

Why Do They Not Abstract?

```
type = ( (extern ? 0x10 : 0)
         | (pcrel ? 0x80 : 0)
         | (neg ? 0x08 : 0)
         | (length << 5));
```

```
type = ( (extern ? 0x08 : 0)
         | (pcrel ? 0x01 : 0)
         | (neg ? 0x10 : 0)
         | (length << 1));
```

Why Do They Not Abstract?

```
if (endianness(abfd) == BIG) {
    type = ( (extern ? 0x10 : 0)
            | (pcrel ? 0x80 : 0)
            | (neg ? 0x08 : 0)
            | (length << 5));
} else {
    type = ( (extern ? 0x08 : 0)
            | (pcrel ? 0x01 : 0)
            | (neg ? 0x10 : 0)
            | (length << 1));
}
```

Why Do They Not Abstract?

```
#define PUT_BITS(endn, size, data, len, shift) \
    ((endn) == BIG ? \
        ((data) & ((1 << (len)) - 1)) << (shift) \
    : ((data) & ((1 << (len)) - 1)) \
        << (size) - (len) - (shift))
```

...

```
end = endianness(abfd);
size = sizeof(type);
type = PUT_BITS(end, size, extern, 1, 4)
      | PUT_BITS(end, size, pcrel, 1, 7)
      | PUT_BITS(end, size, neg, 1, 3)
      | PUT_BITS(end, size, length, 2, 5);
```

Why Do They Not Abstract?

```
if (endianness(abfd) == BIG) {
    type = ( (extern ? 0x10 : 0)
            | (pcrel  ? 0x80 : 0)
            | (neg    ? 0x08 : 0)
            | (length << 5));
} else {
    type = ( (extern ? 0x08 : 0)
            | (pcrel  ? 0x01 : 0)
            | (neg    ? 0x10 : 0)
            | (length << 1));
}
```

```
#define PUT_BITS(endn,size,data,len,shift) \
((endn) == BIG ? \
 ((data)&((1<<(len))-1)) << (shift) \
: ((data)&((1<<(len))-1)) \
 << (size)-(len)-(shift))

...
end = endianness(abfd);
size = sizeof(type);
type = PUT_BITS(end, size, extern, 1, 4)
      | PUT_BITS(end, size, pcrel, 1, 7)
      | PUT_BITS(end, size, neg, 1, 3)
      | PUT_BITS(end, size, length, 2, 5);
```


Our Approach

- We separate bit-level algorithm from data layout.
- Programmer writes
 - code assuming one layout
 - explicit declarative descriptions for how multiple layouts relate to each other

Our Approach

- We separate bit-level algorithm from data layout.
- Programmer writes
 - code assuming one layout
 - explicit declarative descriptions for how multiple layouts relate to each other
- A source-to-source translation generates
 - versions of the code for each layout
 - plain C code, suitable for passing to a C compiler

Our Approach: Example

...

```
char type;
```

...

```
type = ( (extern ? 0x10 : 0)
         | (pcrel  ? 0x80 : 0)
         | (neg    ? 0x08 : 0)
         | (length << 5) );
```

Our Approach: Example

```
enum endian { BIG, LITTLE };
```

...

```
char type;
```

...

```
port (endianness(abfd), BIG) {
```

```
    type = ( (extern ? 0x10 : 0)
```

```
            | (pcrel  ? 0x80 : 0)
```

```
            | (neg    ? 0x08 : 0)
```

```
            | (length << 5));
```

```
}
```

Our Approach: Example

```
enum endian { BIG, LITTLE };
```

```
...
```

```
char type @ match endian, bit with  
          BIG      -> 0:1:2:3:4:5:6:7  
          | LITTLE -> 7:5:6:4:3:2:1:0;
```

```
...
```

```
port (endianness(abfd), BIG) {  
    type = ( (extern ? 0x10 : 0)  
            | (pcrel  ? 0x80 : 0)  
            | (neg    ? 0x08 : 0)  
            | (length << 5));  
}
```

Our Approach: Example

```
enum endian { BIG, LITTLE };
```

```
...
```

```
char type
```

```
@ match endian, bit with
```

```
    BIG      -> 0:1:2:3:4:5:6:7
```

```
    | LITTLE -> 7:5:6:4:3:2:1:0;
```

```
...
```

```
port (endianness(abfd), BIG) {
```

```
    type = ( (extern ? 0x10 : 0)
```

```
            | (pcrel  ? 0x80 : 0)
```

```
            | (neg    ? 0x08 : 0)
```

```
            | (length << 5));
```

```
}
```

Our Approach: Example

```
enum endian { BIG, LITTLE };
```

```
...
```

```
char type @ match endian, bit with  
          BIG      -> 0:1:2:3:4:5:6:7  
          | LITTLE -> 7:5:6:4:3:2:1:0;
```

```
...
```

```
port (endianness(abfd), BIG) {  
    type = ( (extern ? 0x10 : 0)  
            | (pcrel  ? 0x80 : 0)  
            | (neg    ? 0x08 : 0)  
            | (length << 5));  
}
```

Our Approach: Example

```
enum endian { BIG, LITTLE };  
...  
char type @ match endian, bit with  
          BIG      -> 0:1:2:3:4:5:6:7  
          | LITTLE -> 7:5:6:4:3:2:1:0;  
...  
port (endianness(abfd), BIG) {  
    type = ( (extern ? 0x10 : 0)  
            | (pcrel  ? 0x80 : 0)  
            | (neg    ? 0x08 : 0)  
            | (length << 5));  
}
```


Outline

- Motivation
- **Description of Our Tool**
- Experience Applying the Tool
- Conclusion

Tool Components

- Layout annotations on variable and field declarations:
 - to specify how multiple bit-level layouts relate to each other.
- New **port** statement:
 - to delimit bit-level code making layout assumptions.
- Source-to-source translation
 - yields C code handling all specified layouts.

Layout Annotations

```
match endian, bit with
  BIG      -> 0:1:2:3:4:5:6:7
  | LITTLE -> 7:5:6:4:3:2:1:0;
```

Layout Annotations

```
match endian, bit with
```

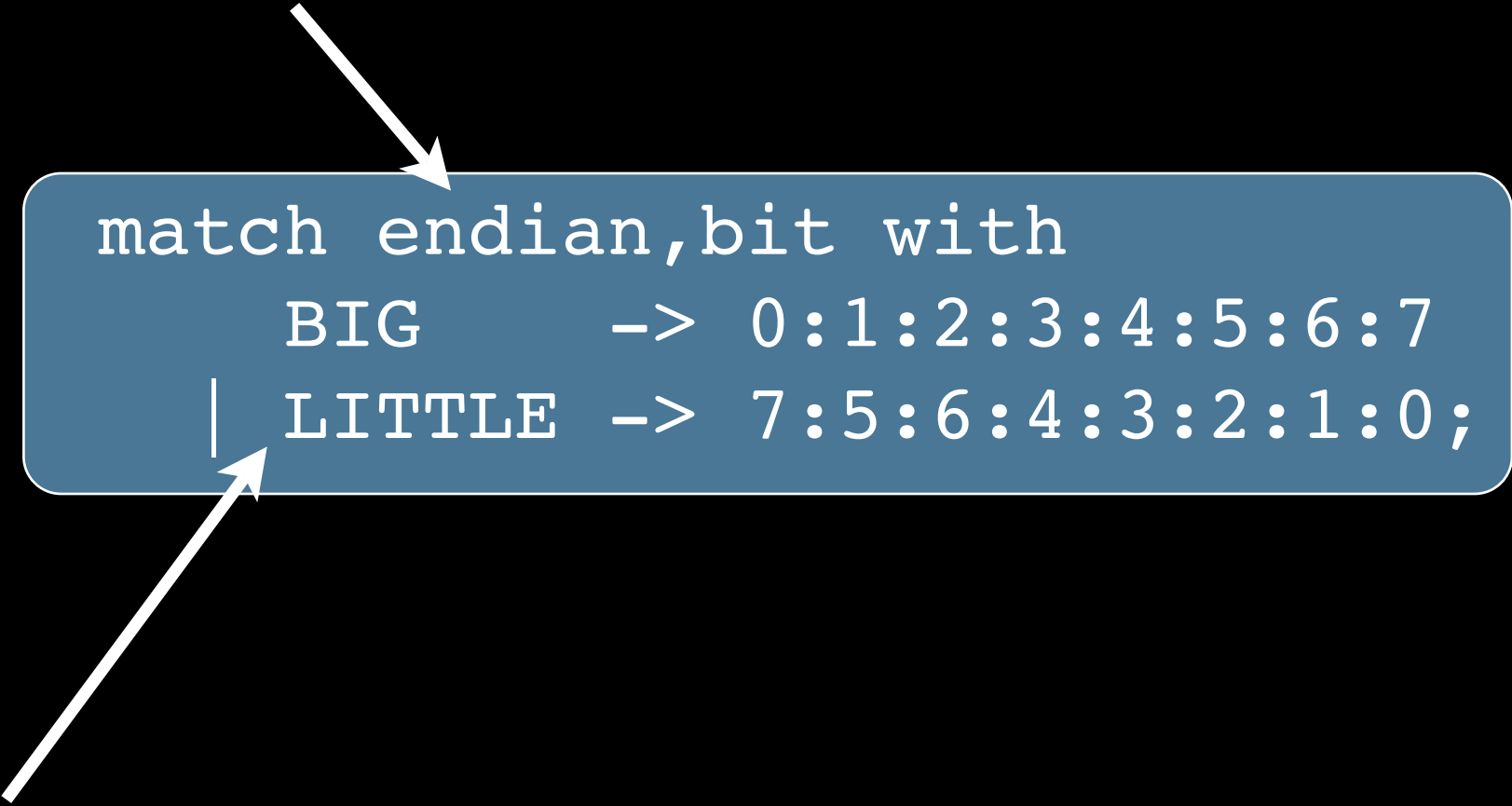
```
    BIG      -> 0:1:2:3:4:5:6:7
```

```
  | LITTLE  -> 7:5:6:4:3:2:1:0;
```

Layout Annotations

Enumeration type

```
enum endian { BIG, LITTLE };
```



```
match endian, bit with  
    BIG      -> 0:1:2:3:4:5:6:7  
| LITTLE    -> 7:5:6:4:3:2:1:0;
```

Enumeration constants

Layout Annotations

Enumeration type

```
enum endian { BIG, LITTLE };
```

Granularity

```
match endian, bit with  
  BIG      -> 0:1:2:3:4:5:6:7  
| LITTLE  -> 7:5:6:4:3:2:1:0;
```

Enumeration constants

Layout Annotations

Enumeration type

```
enum endian { BIG, LITTLE };
```

Granularity

```
match endian, bit with  
  BIG      -> 0:1:2:3:4:5:6:7  
| LITTLE -> 7:5:6:4:3:2:1:0;
```

Enumeration constants

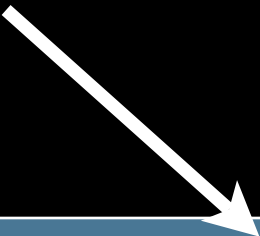
Layouts

Port Statement

```
port (endianness(abfd), BIG) {  
    type = ( (extern ? 0x10 : 0)  
            | (pcrel  ? 0x80 : 0)  
            | (neg    ? 0x08 : 0)  
            | (length << 5));  
}
```


Port Statement

Expression returning
enumeration constant.



```
port (endianness(abfd), BIG) {  
    type = ( (extern ? 0x10 : 0)  
            | (pcrel  ? 0x80 : 0)  
            | (neg    ? 0x08 : 0)  
            | (length << 5));  
}
```

Port Statement

Expression returning
enumeration constant.

Enumeration constant.

```
port (endianness(abfd), BIG) {  
    type = ( (extern ? 0x10 : 0)  
            | (pcrel  ? 0x80 : 0)  
            | (neg    ? 0x08 : 0)  
            | (length << 5));  
}
```

Port Statement

Expression returning
enumeration constant.

Enumeration constant.

```
port (endianness(abfd), BIG) {  
    type = ( (extern ? 0x10 : 0)  
            | (pcrel  ? 0x80 : 0)  
            | (neg    ? 0x08 : 0)  
            | (length << 5));  
}
```

Code written assuming
`endianness(abfd) == BIG`

Translation

- Erases layout annotations.

Translation

- Erases layout annotations.
- Generates “*flip*” and “*unflip*” functions.
 - *Flip* transforms data to a different layout.
 - *Unflip* modifies flipped data to its original form.

Translation

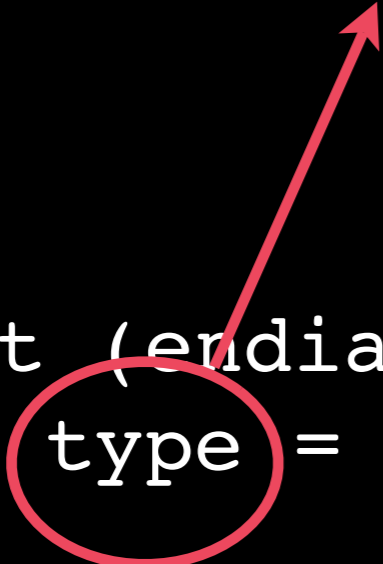
- Erases layout annotations.
- Generates “*flip*” and “*unflip*” functions.
 - *Flip* transforms data to a different layout.
 - *Unflip* modifies flipped data to its original form.
- Translates *port* blocks to C code such that
 - *Flip* functions are called if needed.
 - Code is run.
 - *Unflip* functions are called on flipped data.

Translation Example

```
enum endian { BIG, LITTLE };  
...  
char type @ match endian, bit with  
          BIG      -> 0:1:2:3:4:5:6:7  
          | LITTLE -> 7:5:6:4:3:2:1:0;  
...  
port (endianness(abfd), BIG) {  
    type = ( (extern ? 0x10 : 0)  
            | (pcrel  ? 0x80 : 0)  
            | (neg    ? 0x08 : 0)  
            | (length << 5));  
}
```

Translation Example

```
enum endian { BIG, LITTLE };  
...  
char type @ match endian, bit with  
          BIG    -> 0:1:2:3:4:5:6:7  
          | LITTLE -> 7:5:6:4:3:2:1:0;  
...  
port (endianness(abfd), BIG) {  
    type = ( (extern ? 0x10 : 0)  
            | (pcrel  ? 0x80 : 0)  
            | (neg    ? 0x08 : 0)  
            | (length << 5));  
}
```



Translation Example

```
...
int _tmp = endianness(abfd);
switch(_tmp) {
case LITTLE: flip(& type); break;
case BIG:    break;
}

type = ( (extern ? 0x10 : 0)
        | (pcrel  ? 0x80 : 0)
        | (neg    ? 0x08 : 0)
        | (length << 5));

switch(_tmp) {
case LITTLE: unflip(& type); break;
case BIG:    break;
}
```

Translation Example

```
...
int _tmp = endianness(abfd);
switch(_tmp) {
case LITTLE: flip(& type); break;
case BIG:    break;
}

```

```
type = ( (extern ? 0x10 : 0)
        | (pcrel  ? 0x80 : 0)
        | (neg    ? 0x08 : 0)
        | (length << 5));

```

```
switch(_tmp) {
case LITTLE: unflip(& type); break;
case BIG:    break;
}

```

Translation Example

```
...  
int _tmp = endianness(abfd);  
switch(_tmp) {  
case LITTLE: flip(& type); break;  
case BIG: break;  
}
```

```
type = ( (extern ? 0x10 : 0)  
        | (pcrel ? 0x80 : 0)  
        | (neg ? 0x08 : 0)  
        | (length << 5));
```

```
switch(_tmp) {  
case LITTLE: unflip(& type); break;  
case BIG: break;  
}
```

Translation Example

```
...
int _tmp = endianness(abfd);
switch(_tmp) {
case LITTLE: flip(& type); break;
case BIG:    break;
}
```

```
type = ( (extern ? 0x10 : 0)
        | (pcrel  ? 0x80 : 0)
        | (neg    ? 0x08 : 0)
        | (length << 5));
```

```
switch(_tmp) {
case LITTLE: unflip(& type); break;
case BIG:    break;
}
```

Flip/Unflip Example

```
match endian, bit with
    BIG      -> 0:1:2:3:4:5:6:7
    | LITTLE -> 7:5:6:4:3:2:1:0;
```

```
void flip(void * input) {
    char *t0 = (char*)input;
    char t1 = t0[0];
    t0[0] = 0;
    t0[0] |= ((t1 << 7) & 0x80);
    t0[0] |= ((t1 << 4) & 0x40);
    t0[0] |= ((t1 << 4) & 0x20);
    t0[0] |= ((t1 << 1) & 0x10);
    t0[0] |= ((t1 << 1) & 0x08);
    t0[0] |= ((t1 << 3) & 0x04);
    t0[0] |= ((t1 << 5) & 0x02);
    t0[0] |= ((t1 << 7) & 0x01);
}
```

```
void unflip(void * input) {
    char *t0 = (char*)input;
    char t1 = t0[0];
    t0[0] = 0;
    t0[0] |= ((t1 << 7) & 0x80);
    t0[0] |= ((t1 << 5) & 0x40);
    t0[0] |= ((t1 << 3) & 0x20);
    t0[0] |= ((t1 << 1) & 0x10);
    t0[0] |= ((t1 << 1) & 0x08);
    t0[0] |= ((t1 << 4) & 0x04);
    t0[0] |= ((t1 << 4) & 0x02);
    t0[0] |= ((t1 << 7) & 0x01);
}
```

Summary

- We use compiler technology to do what it does well:
 - Equivalent versions.
 - Low-level code from high-level descriptions.
 - Automation.

Outline

- Motivation
- Description of Our Tool
- **Experience Applying the Tool**
- Conclusion

Overview

- Preliminary experiences suggests that our tool
 - improves readability
 - shrinks the code base
 - minimizes development and maintenance issues
 - applies to the vast majority of occurrences
layout-dependent bit-level code

Experience with GDB/BFD

- GDB/BFD:
 - 1 million lines of code
 - 1700 C files
- We manually inspected 120 files.
 - 407 occurrences of duplication.
 - 3600 lines of duplicated code.

Experience with GDB/BFD

- We applied our tool to 10 files.
 - 31 occurrences of duplicated code
- Our tool applied to 29 of them.
- Eliminated 188 lines (2465 lexical tokens)
- 11 layout annotations required.

Performance

- Byte-level flips optimized away by **gcc -O3**, e.g.:

```
flip(x); y=x[0]; unflip(x);    ➔    y=x[3];
```

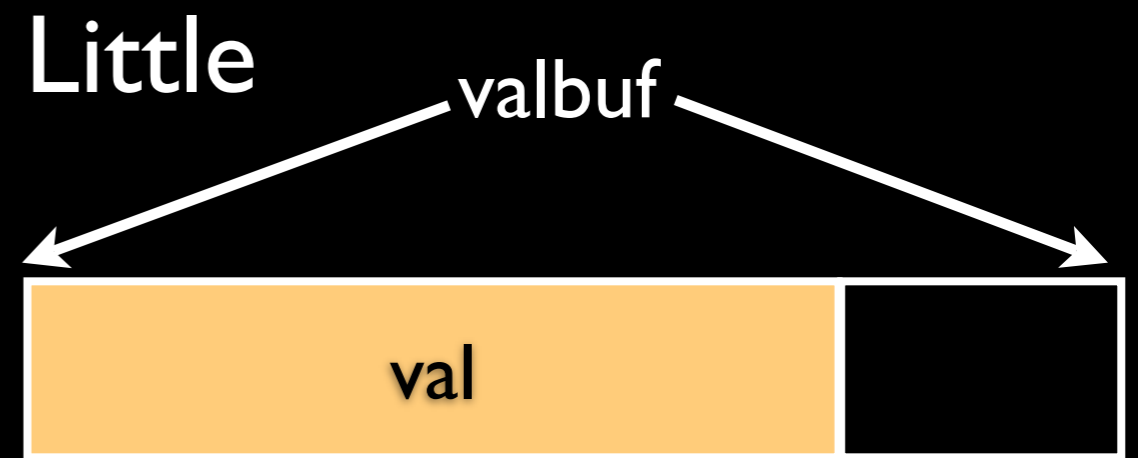
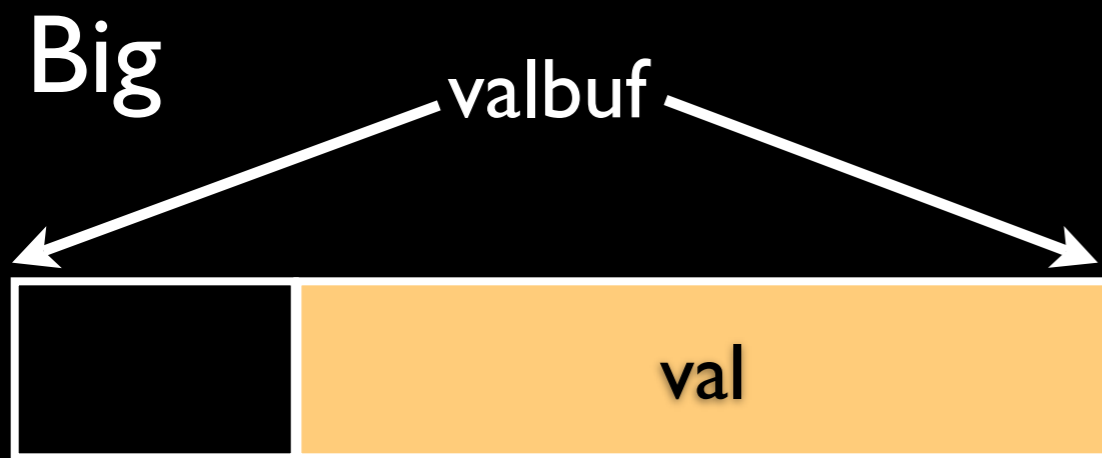
- Bit-level flips can add 50%-100% overhead.
- Layout-dependent code is usually not performance-critical. (E.g., processing file headers.)

An Exception

```
char valbuf[4];  
...  
if (TARGET_BYTE_ORDER == BIG)  
    memcpy(valbuf + (4 - len), val, len);  
else  
    memcpy(valbuf, val, len)
```

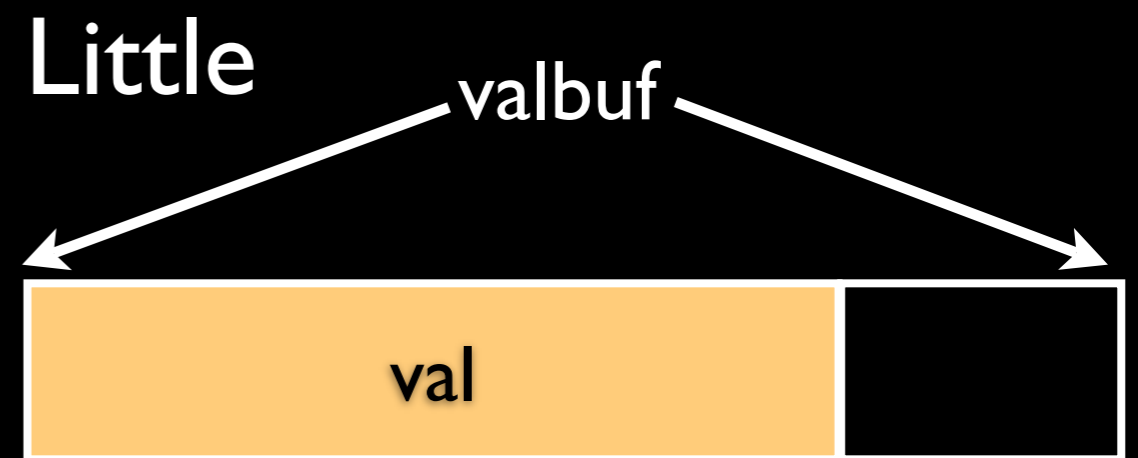
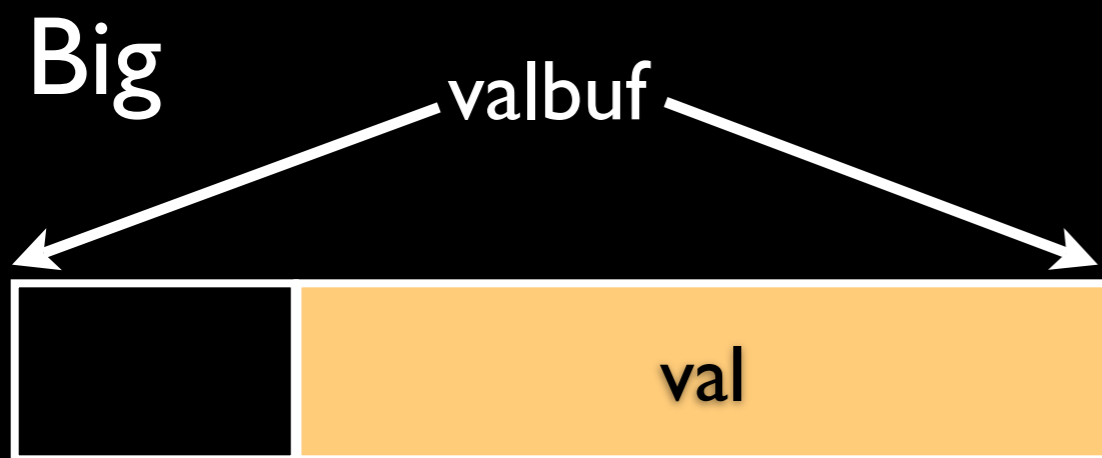
An Exception

```
char valbuf[4];  
...  
if (TARGET_BYTE_ORDER == BIG)  
    memcpy(valbuf + (4 - len), val, len);  
else  
    memcpy(valbuf, val, len)
```



An Exception

```
char valbuf[4];  
...  
if (TARGET_BYTE_ORDER == BIG)  
    memcpy(valbuf + (4 - len), val, len);  
else  
    memcpy(valbuf, val, len)
```



Conclusions

- Lightweight tool to ease writing and maintaining portable bit-level C code.
- Separates bit-level algorithm from layout concerns.
- Layout descriptions provide explicit documentation.
- Performance impact is small.
- Can be easily integrated in the development process:
 - Generated C code is shipped.
 - Users of the source base do not need our tool.

Future Work

- Annotation inference.
- Support for inequivalent data layouts.
 - For example, 32- vs 64-bit words.
- More sophisticated transformations.
 - Transform code rather than data.

Thank You!

wasp.cs.washington.edu