REVERSE
ENGINEER

TODO

- Assignment 3 assigned due October 28

gdb exercises

Overview

- Switch statements

Structured programming constructs

- ~~`if`~~
- ~~`do`~~
- ~~`while`~~
- ~~`for`~~
- `switch`

Switch statements

- How do we convert these to x86 assembly?

```
void print_digit(int digit) {
  switch (digit) {
    case 0:
      printf("zero\n");
      break;
    case 1:
      printf("one\n");
      break;
    case 2:
      printf("two\n");
      break;
    case 3:
      printf("three\n");
      break;
    ...
    case 9:
      printf("nine\n");
      break;
    default:
      printf("not a digit\n");
      break;
  }
}
```

Anatomy of a switch statement

- `switch(...)` – the value to be compared
- `case …` – the value to compare it against
- `default` – the catch-all case for unmatched values
  - if no `default` case, the default is to do nothing
- `break` – case is finished, break out of switch

```
void print_digit(int digit) {
  switch (digit) {
    case 0:
      printf(“zero\n”);
      break;
    case 1:
      printf(“one\n”);
      break;
    case 2:
      printf(“two\n”);
      break;
    case 3:
      printf(“three\n”);
      break;
    ...
    case 9:
      printf(“nine\n”);
      break;
    default:
      printf(“not a digit\n”);
      break;
  }
}
```

```
void print_digit(int digit) {
    if (digit == 0) {
        printf("zero\n");
    } else if (digit == 1) {
        printf("one\n");
    } else if (digit == 2) {
        printf("two\n");
    } else if (digit == 3) {
        printf("three\n");
    } ... else if (digit == 9) {
        printf("nine\n");
    } else {
        printf("not a digit\n");
    }
}
```

switch → if

- Is there any downside to the “if” approach?
- What if there are 1,000 cases?
- There is another problem though
  - What if there is no break?

```
long switch_example
      (long x, long y, long z){
    long w = 1;
    switch(x) {
       case 1:
          w = y*z;
          break;
       case 2:
          w = y/z;
       case 3:
          w += z;
          break;
       case 5:
       case 6:
          w -= z;
          break;
       default:
          w = 2;
    }
    return w;
}
```



No break (fall through)



No case 4 (default)



Fall through (case 5 same as 6)

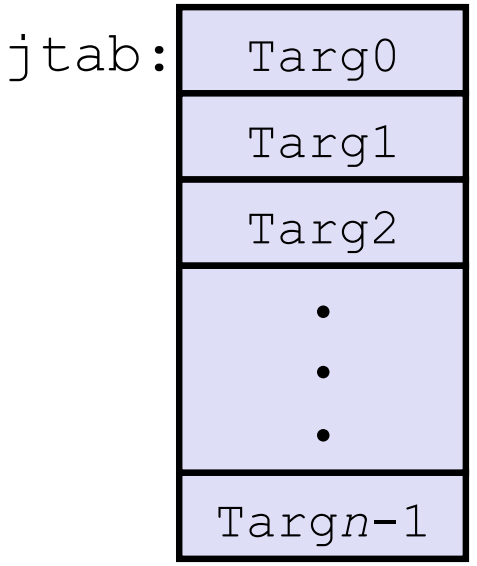
A table to the rescue

- A jump table – table where each index is the address of the code for one of the cases
- Index 0 is the address of the code for case 0
- Index 1 is the address of the code for case 1
- …
- Index *n*-1 is the address of the code for case *n*-1

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table



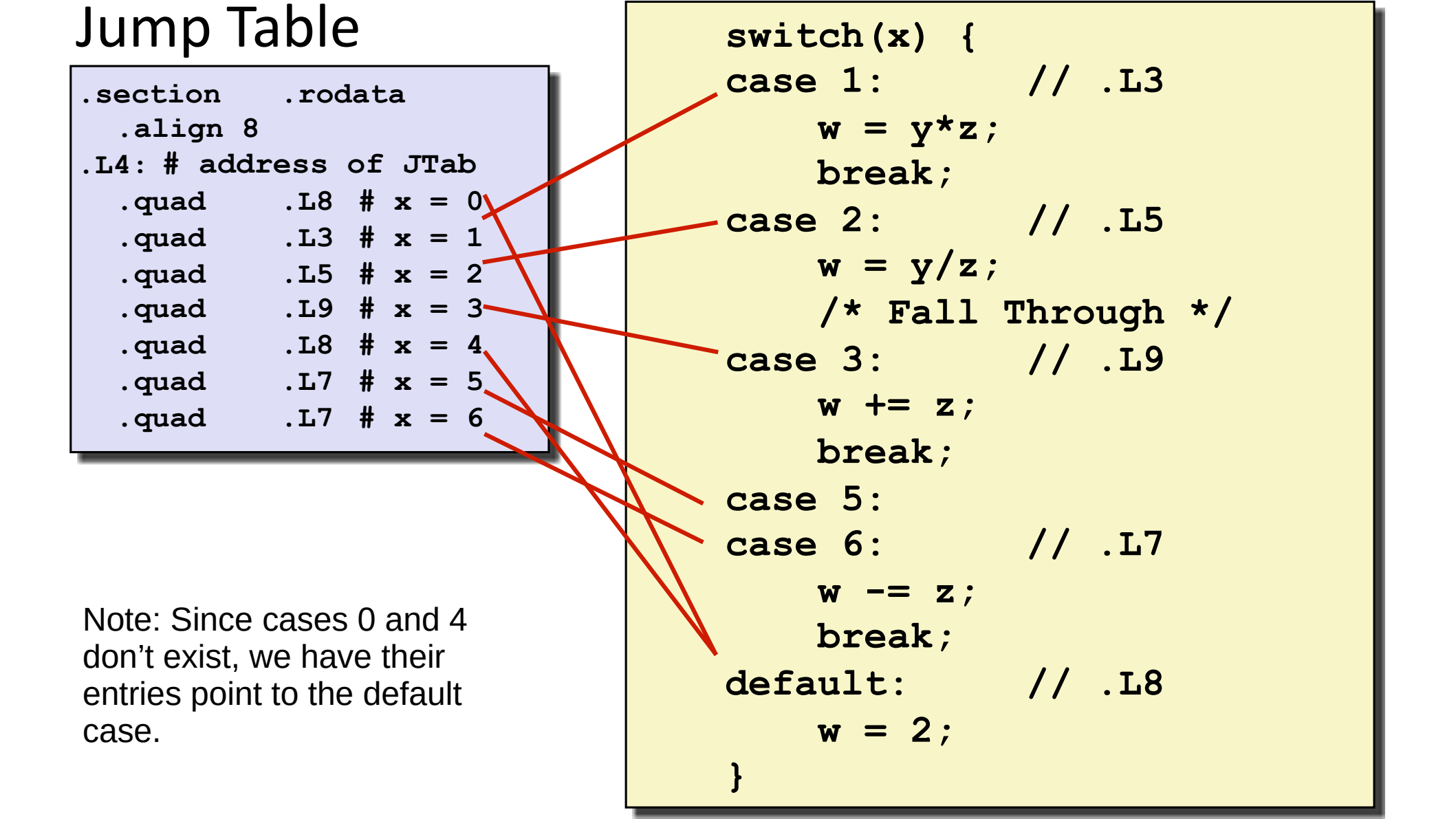
Translation (Extended C)

```
goto *jtab[x];
```

Jump Table

```
.section    .rodata
  .align 8
.L4: # address of JTab
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L5  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L7  # x = 5
  .quad     .L7  # x = 6
```

```
switch(x) {
case 1:         // .L3
    w = y*z;
    break;
case 2:         // .L5
    w = y/z;
    /* Fall Through */
case 3:         // .L9
    w += z;
    break;
case 5:
case 6:         // .L7
    w -= z;
    break;
default:        // .L8
    w = 2;
}
```



Note: Since cases 0 and 4 don't exist, we have their entries point to the default case.

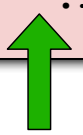
```
long switch_example
    (long x, long y, long z){
  long w = 1;
  switch(x) {
    case 1:
      w = y*z;
      break;
    case 2:
      w = y/z;
      /* Fall through */
    case 3:
      w += z;
      break;
    case 5:
    case 6:
      w -= z;
      break;
    default:
      w = 2;
  }
  return w;
}
```

Jump Table

```
.L4: # address of JTab
  .quad   .L8   # x = 0
  .quad   .L3   # x = 1
  .quad   .L5   # x = 2
  .quad   .L9   # x = 3
  .quad   .L8   # x = 4
  .quad   .L7   # x = 5
  .quad   .L7   # x = 6
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

```
switch_example:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```



Indirect jump

Indirect jump

- A new kind of jump
- Previously seen:
  - Conditional (`jle`) vs. unconditional jump (`jmp`)
- But there is another axis
- Direct vs. indirect jump:
  - Direct – jumps to a fixed address (`jmp malloc`)
  - Indirect – jumps to a computed address (`jmp *%rax`)

So that’s how we handle the cases

- But what about the default case?
- “Missing” cases (`x == 0` or `x == 4`) point to the default case, but how do we handle `x == 10000`?
- Does our table need 10000 entries?

```
long switch_example
    (long x, long y, long z){
  long w = 1;
  switch(x) {
    case 1:
      w = y*z;
      break;
    case 2:
      w = y/z;
      /* Fall through */
    case 3:
      w += z;
      break;
    case 5:
    case 6:
      w -= z;
      break;
    default:
      w = 2;
  }
  return w;
}
```

Jump Table

```
.L4: # address of JTab
  .quad  .L8  # x = 0
  .quad  .L3  # x = 1
  .quad  .L5  # x = 2
  .quad  .L9  # x = 3
  .quad  .L8  # x = 4
  .quad  .L7  # x = 5
  .quad  .L7  # x = 6
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

```
switch_example:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```



Switch guard

Guarding the switch

- A range check to make sure that our switched value is within the bounds of the jump table
- If not, jump to the default case
  - Do not use the jump table at all

```
long w = 1;
switch(x) {
case 1:       // .L3
    w = y*z;
    break;
case 2:       // .L5
    w = y/z;
    /* Fall Through */
case 3:       // .L9
    w += z;
    break;
case 5:
case 6:       // .L7
    w -= z;
    break;
default:      // .L8
    w = 2;
}
return w;
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	Return value

```
switch_example:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8) # goto *JTab[x]
```

```
.L3:                        # Case 1
   movq    %rsi, %rax       # y
   imulq   %rdx, %rax       # y*z
   ret
.L5:                        # Case 2
   movq    %rsi, %rax
   cqto
   idivq   %rcx             #  y/z
   jmp     .L6              #  goto merge
.L9:                        # Case 3
   movl    $1, %eax         #  w = 1
.L6:                        # merge:
   addq    %rcx, %rax       #  w += z
   ret
.L7:                        # Case 5,6
  movl  $1, %eax            #  w = 1
  subq  %rdx, %rax          #  w -= z
  ret
.L8:                        # Default:
  movl  $2, %eax            #  2
  ret
```

Guarding the switch

- Handles switched values being out of range
- What if the cases themselves are in a crazy range?
- `case 1:`
- `case 100:`
- `case 10000:`
- Does our jump table need 10000 entries?

There is an art to this

- Compiler will only use a jump table if cases are sufficiently compact
- case 0, case 1243234, case 0xdeadbeef – jump table would be too large
- What about case 1000, 1001, 1002, 1003?
- What about case 0, 0x10000, 0x20000, 0x30000?

switch*.c

Jump table summary

- Pros:
  - Can handle fall-through easily
  - Evaluate against a gazillion cases in constant time
- Cons:
  - Require “dense” cases (although the compiler can help with making them dense)
  - Can only switch on integers

Fallback

- If a jump table is not feasible:
- `if (x == case #1) goto code block 1`
- `if (x == case #2) goto code block 2`
- `...`
- `[code block 1]`
- `[code block 2]`
- `...`

Duff’s device

```
void duffs_device(char *to, char *from, unsigned long count) {
    if (count == 0) return;

    unsigned long n = count / 8 + (count % 8 != 0);

    switch (count % 8) {
    case 0: do { *to++ = *from++;
    case 7:      *to++ = *from++;
    case 6:      *to++ = *from++;
    case 5:      *to++ = *from++;
    case 4:      *to++ = *from++;
    case 3:      *to++ = *from++;
    case 2:      *to++ = *from++;
    case 1:      *to++ = *from++;
            } while (--n);
    }
}
```

Switch exercises

- Recall that 'A' is syntactic sugar for 0x41, etc.:
  - 'A' - 'A' → 0
  - 'B' - 'A' → 1
  - 'C' - 'A' → 2
  - 'B' - 'B' → 0
  - 'C' - 'B' → 1
  - …

- Recall that in C you can say:

```
switch (x) {
case 'A':
   ...
case 'B':
   ...
}
```

Next up: function calls

- We’ve seen a glimpse
- But how are stack variables automatically allocated?
- How does a `ret` instruction “know” where to return? How does it know where it was called from?

TODO

- Assignment 3 assigned due October 28