Google

Switch Lowering in LLVM

EDAN75
8 October 2018
hwennborg (at) google.com

Why study optimizing compilers?

- Teaches what the compiler can do for you
- Need to look at the output of the compiler!

Arithmetic sum

```
int sum_to(unsigned x) {
    int sum = 0;
    for (int i = 0; i <= x; i++) {
        sum += i;
    }
    return sum;
}
```

```
0 + 1 + 2 + … + x
```

```
$ clang -O2 -S -o - a.cc
```

Some kind of loop

```
int f(unsigned x) {
    int n = 0;
    while (x) {
        x &= x - 1;
        n++;
    }
    return n;
}

$ clang -O2 -march=native -S -o - b.cc
```

Making a choice

```
void f(); void g();

void foo(unsigned x) {
    if (x == 0 || x == 3 || x == 14 || x == 17 || x == 18 || x == 29) {
        f();
    } else {
        g();
    }
}

$ clang -O2 -S -o - c.cc
```

What is a switch statement?

```
void g(int);

void f(int x) {
    switch (x) {
    case 0:  g(1); break;
    case 1:  g(3); break;
    case 42: g(0); break;
    }
}

$ clang -O2 -g0 -emit-llvm -S -o - d.cc
```

Is this also a switch statement?

```
void g(int);

void f(int x) {
    if (x == 2) {
        g(1);
    } else if (x == 5) {
        g(2);
    } else if (x == 52) {
        g(0);
    }
}
```

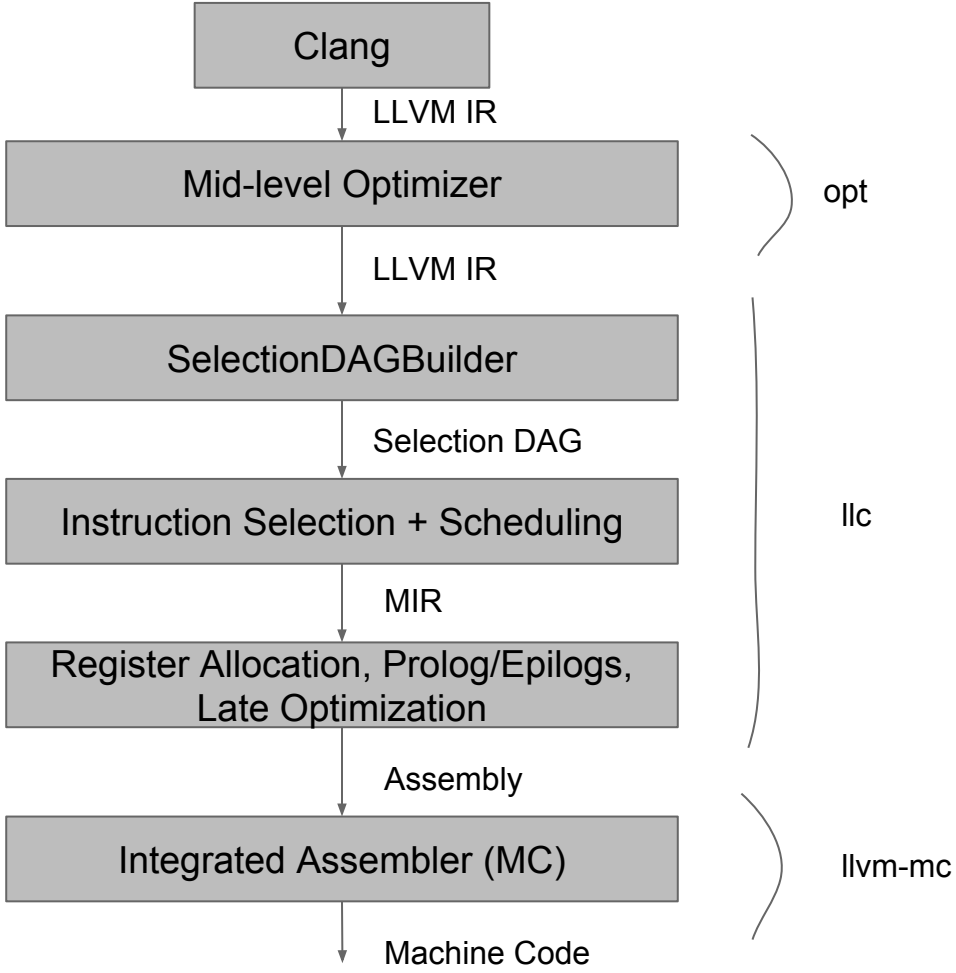
```
$ clang -O2 -g0 -emit-llvm -S -o - x.cc
```

LLVM’s optimizer makes it a switch statement.

```
void g(int);

void f(int x) {
    if (x == 2) {
        g(1);
    } else if (x == 5) {
        g(2);
    } else if (x == 52) {
        g(0);
    }
}
```

```
$ clang -O0 -g0 -emit-llvm -S -o - x.cc
```


Clang
LLVM IR
Mid-level Optimizer
opt
LLVM IR
SelectionDAGBuilder
Selection DAG
Instruction Selection + Scheduling
llc
MIR
Register Allocation, Prolog/Epilogs,
Late Optimization
Assembly
Integrated Assembler (MC)
llvm-mc
Machine Code
Object file

Looking at each stage of the process

```
Generating LLVM IR with Clang:
$ clang -O2 -g0 -Xclang -disable-llvm-optzns -emit-llvm -S -o x.ll x.cc

Running the mid-level optimizers:
$ opt -O3 -S -o x.opt.ll x.ll -print-after-all

Running the code generator:  (“apt-get install xdot” to view DAGs)
$ llc -filetype=asm -o x.s x.opt.ll -view-isel-dags -print-after-all

Running the assembler:
$ llvm-mc -filetype=obj -o x.o x.s

Looking at the object file:
$ objdump -dr x.o
```

IR-Level Switch Optimizations: “Select Switches”

```
int f(int x) {
    switch (x) {
    case 0: return 42;
    case 1: return 52;
    case 2: return 62;
    case 3: return 72;
    case 4: return 82;
    }
}
```

```
$ clang -O2 -g0 -emit-llvm -S -o - e.cc
```

See https://github.com/llvm-mirror/llvm/blob/release_60/lib/Transforms/Utils/SimplifyCFG.cpp#L4935

IR-Level Switch Optimizations: “Select Switches”

```
int f(int x) {
    switch (x) {
    case 0: return 42;
    case 1: return 19;
    case 2: return 12;
    case 3: return 17;
    case 4: return 72;
    }
}
```

```
$ clang -O2 -g0 -emit-llvm -S -o - f.cc
```

See SwitchToLookupTable() in SimplifyCFG.cpp

IR-Level Switch Optimizations: “Select Switches”

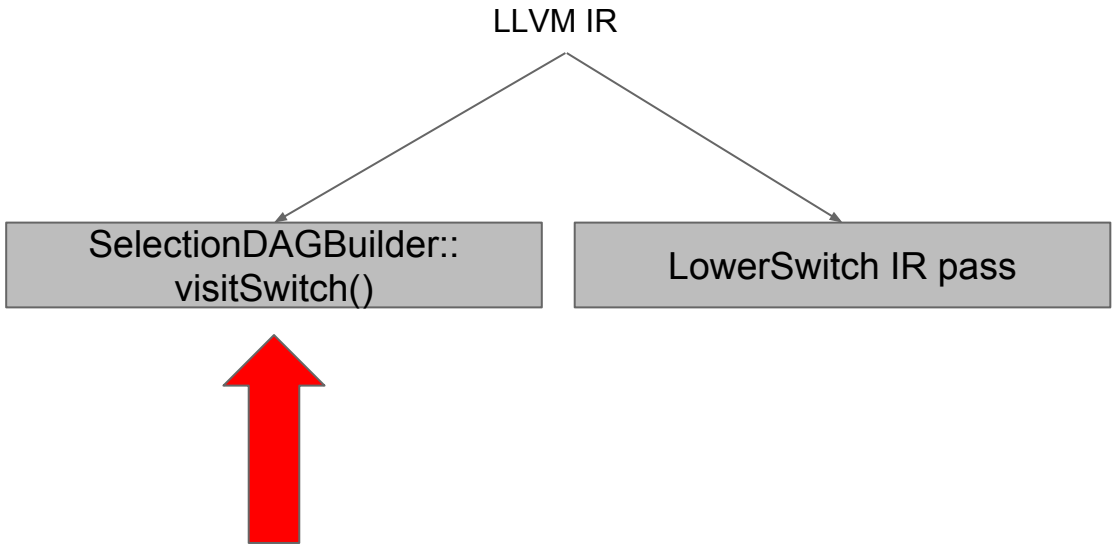
```
bool f(int x) {
    switch (x) {
    case 0: return true;
    case 1: return true;
    case 2: return false;
    case 3: return false;
    case 4: return true;
    }
}
```

```
$ clang -O2 -g0 -emit-llvm -S -o - g.cc
```

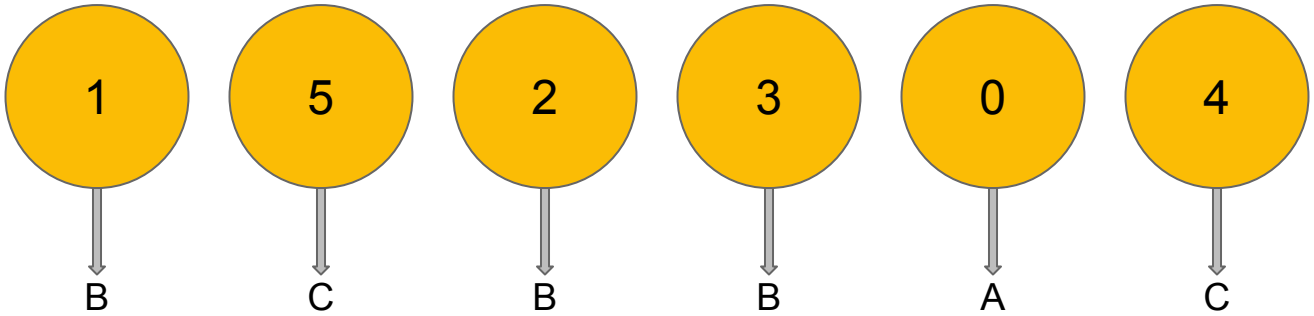
See https://github.com/llvm-mirror/llvm/blob/release_60/lib/Transforms/Utils/SimplifyCFG.cpp#L4972

General Switch Lowering

```
void f(int x) {
    switch (x) {
    case 0:  // Stuff.
    case 1:  // More stuff.
    case 42: // Other stuff.
    }
}
```

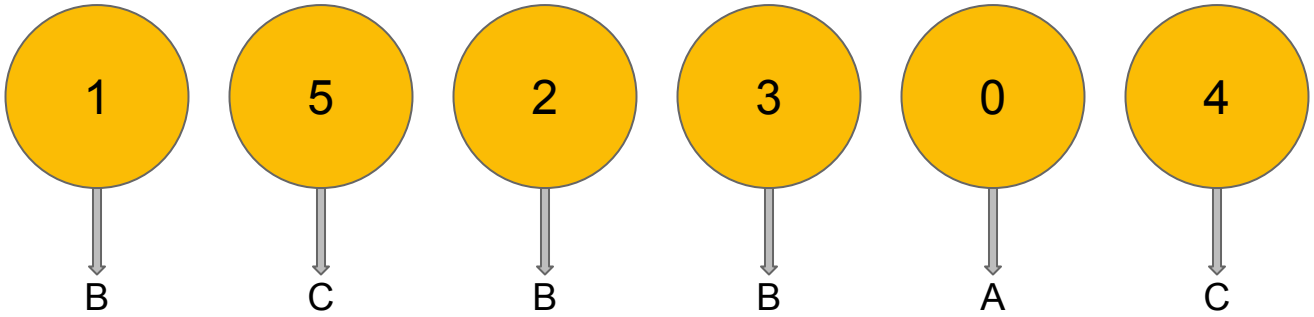


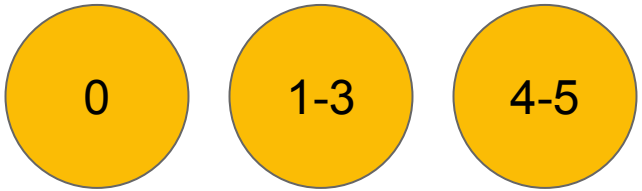
Step 0: Cluster adjacent cases




```
void f(int x) {
  switch (x) {
A:  case 0: // Stuff.
B:  case 1: case 2: case 3: // More stuff.
C:  case 4: case 5: // Other stuff.
  }
}
```

Step 0: Cluster adjacent cases

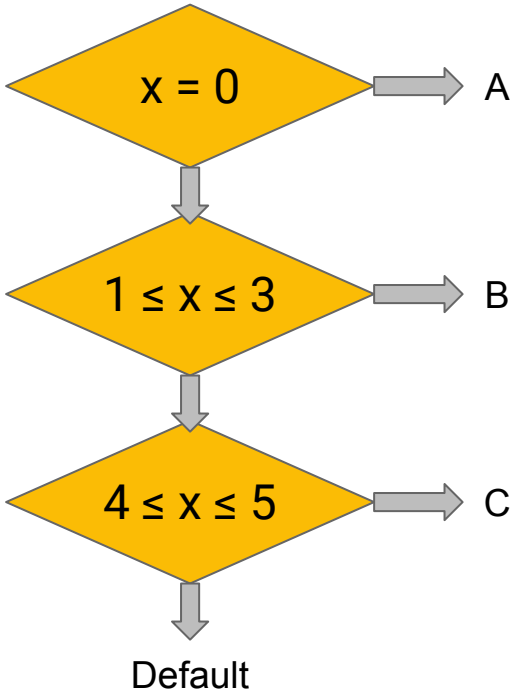




Lowering strategies

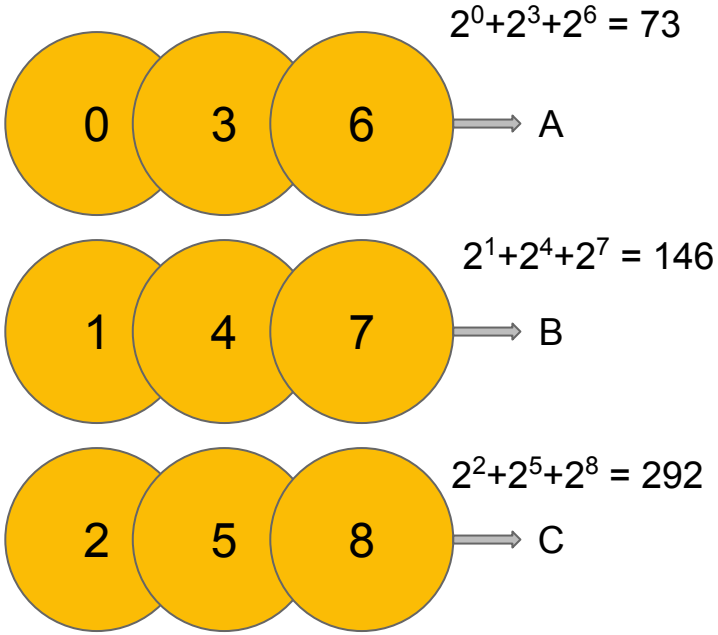
1. Straight comparisons
2. Bit tests
3. Jump table
4. Binary search tree

1. Straight comparisons

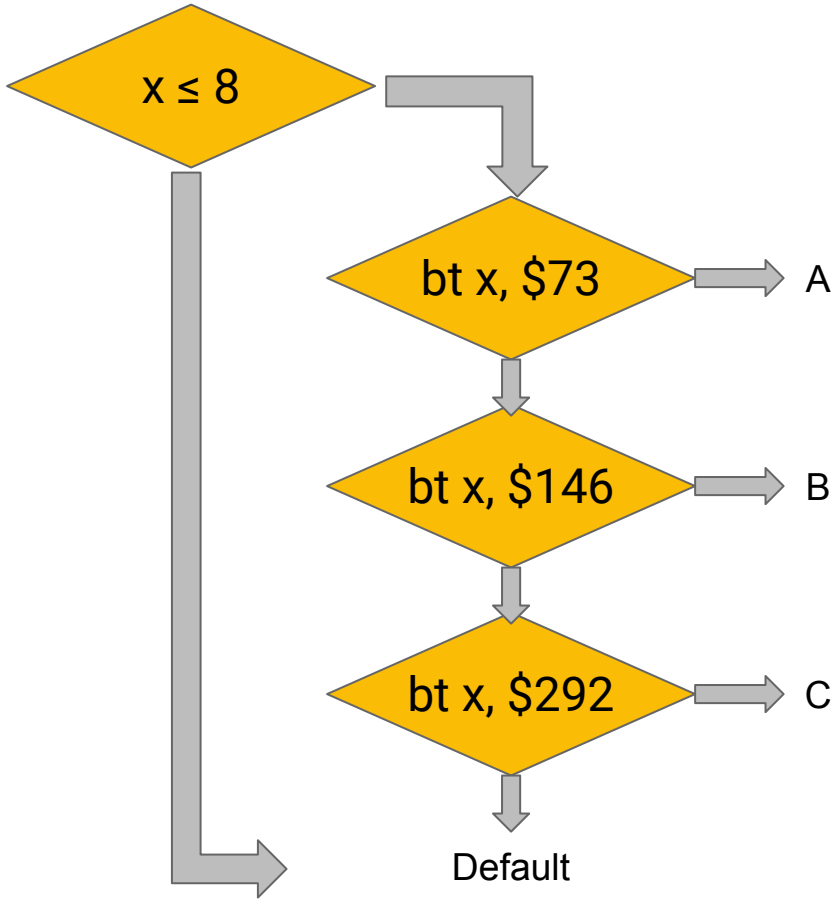


- Number of clusters ≤ 3

2. Bit tests



- Number of destinations ≤ 3
- Range fits in machine word



3. Jump table

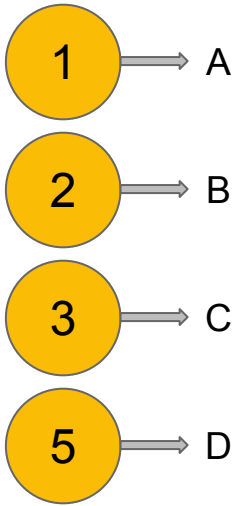
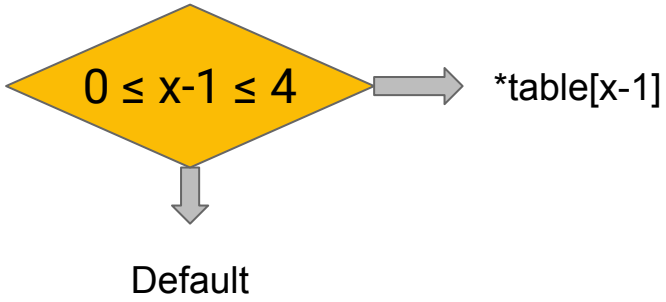


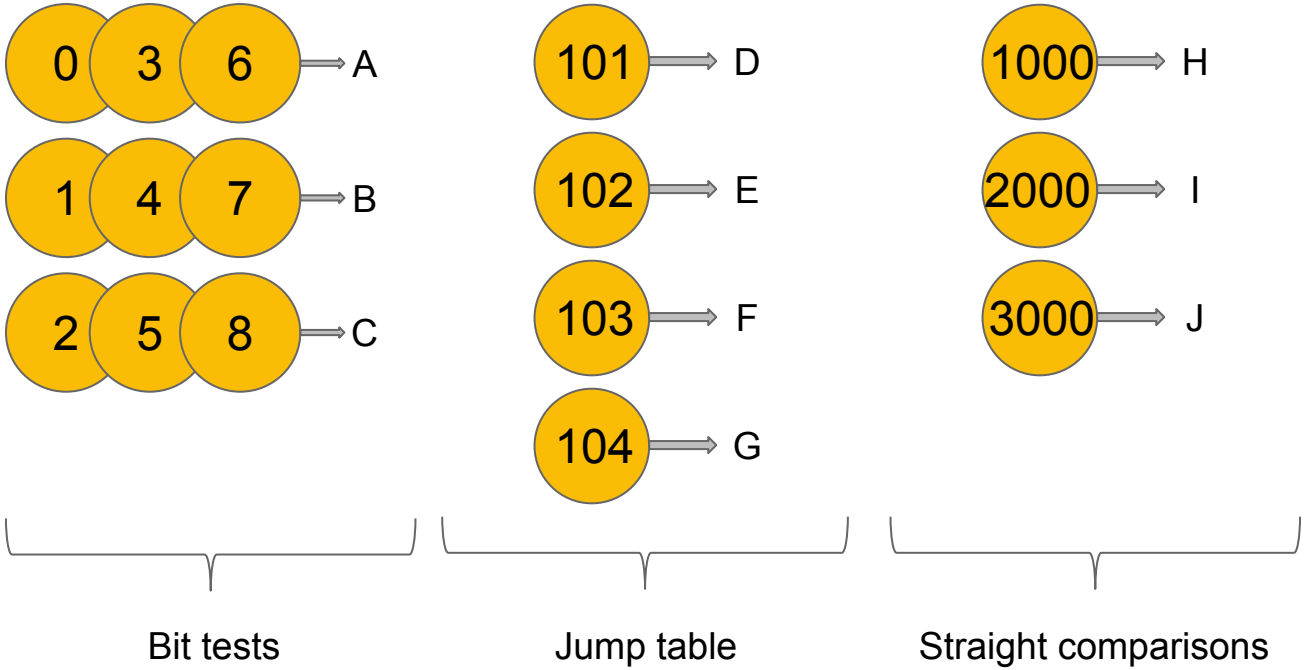
table:

0	A
1	B
2	C
3	Default
4	D

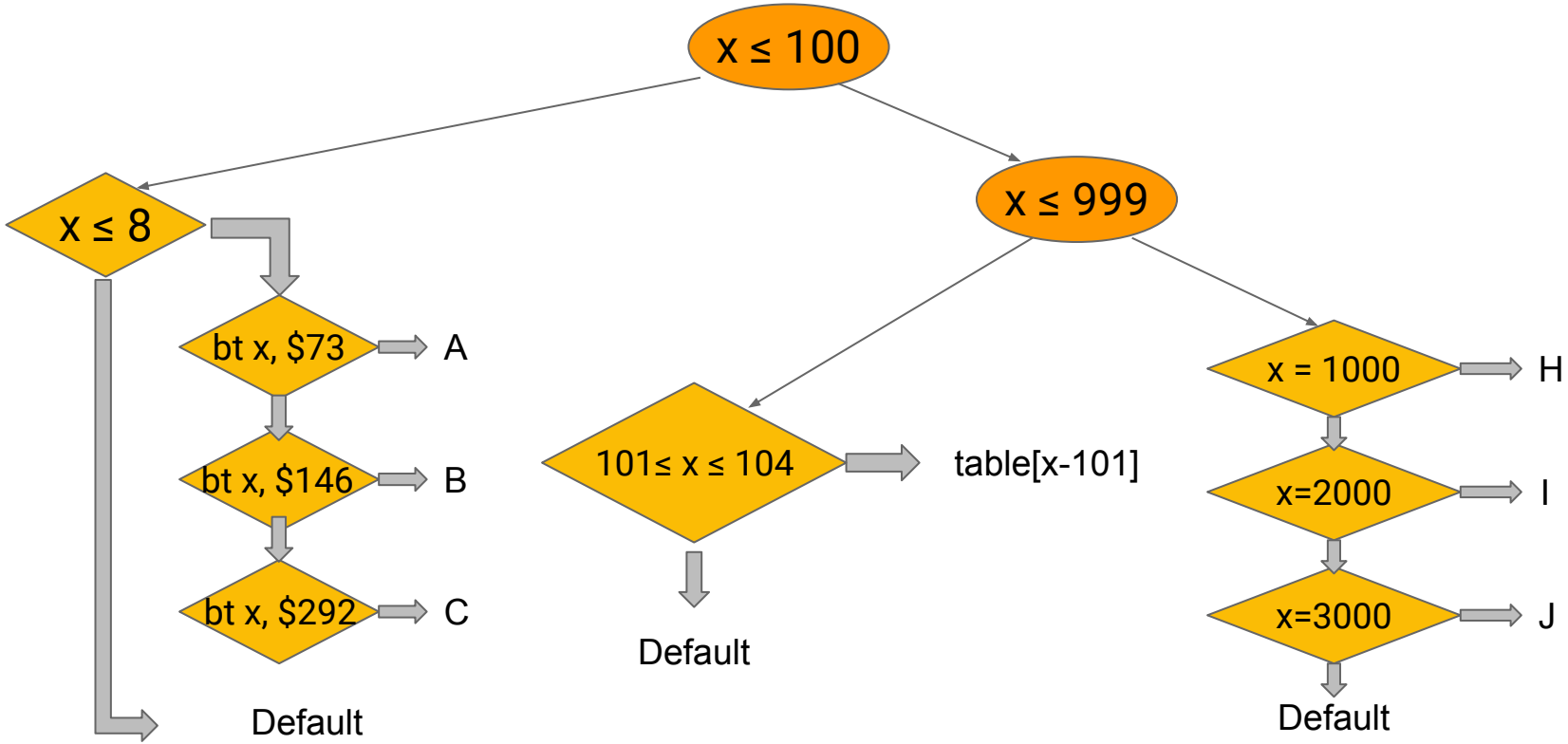


- Number of clusters ≥ 4
- Table density ≥ 10% (or 40% for -Os)

4. Binary search tree



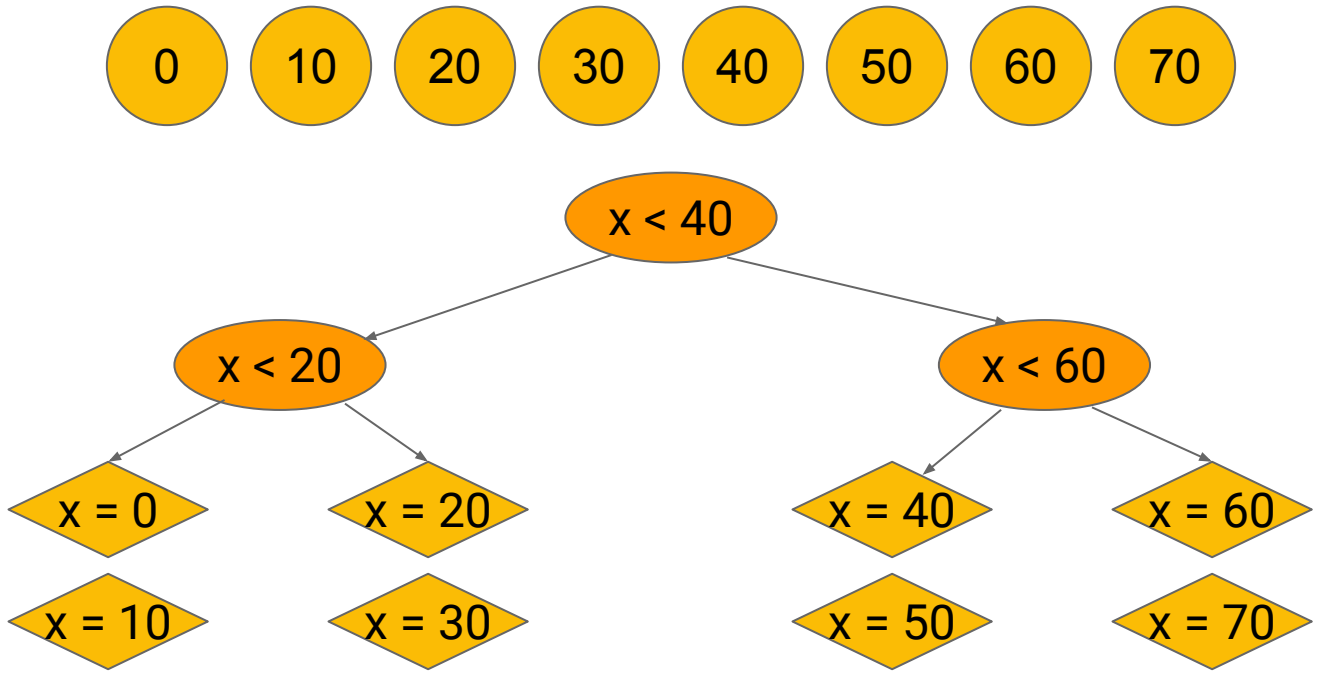
4. Binary search tree



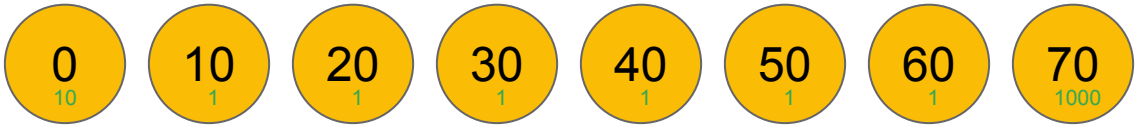
Bottom-up tree construction

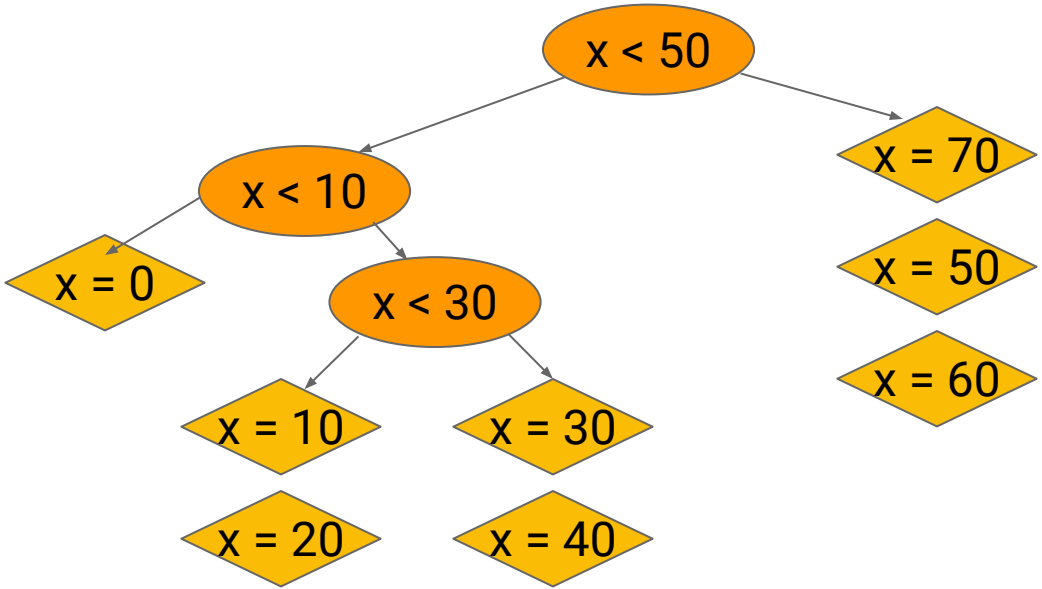
- Consider the whole range of cases
- Find case clusters suitable for bit tests
- Find case clusters suitable for jump tables
- Build binary search tree

Balanced by node count

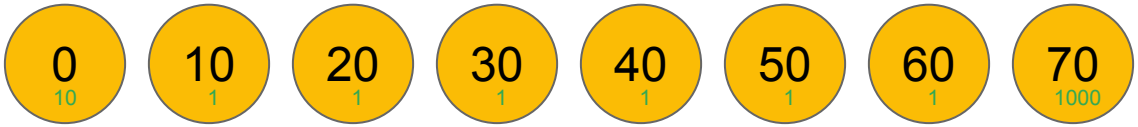


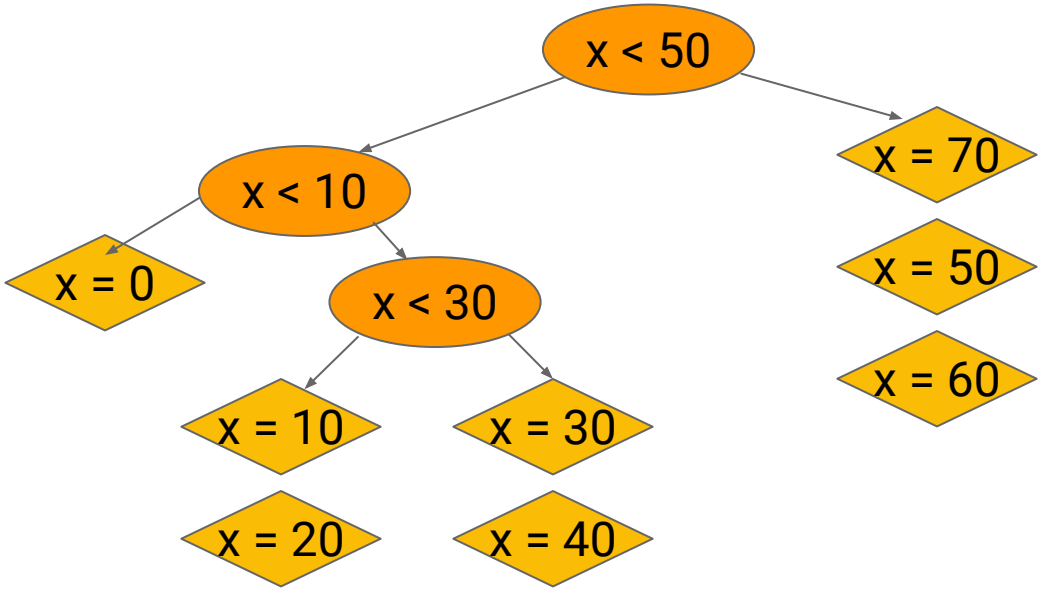
Balanced by node weight





Balanced by node weight





x	Branches	x weight
0	3	30
10	4	4
20	5	5
30	4	4
40	5	5
50	3	3
60	4	4
70	2	2000

(Without weight balancing: 3052) Sum: 2055

Takeaways

- The compiler can do a lot for you
- Look at the output (use godbolt.org to try multiple compilers)
- Fairly easy to follow code through LLVM