Lecture 15 — Control-Flow Analysis

David J. Pearce

School of Engineering and Computer Science
Victoria University of Wellington

Control-Flow Analysis (CFA)

> *In computer science,* **control flow analysis (CFA)** *is a static code analysis technique for determining the control flow of a program.*
>
> *–Wikipedia*

How to perform control-flow analysis for AVR programs?

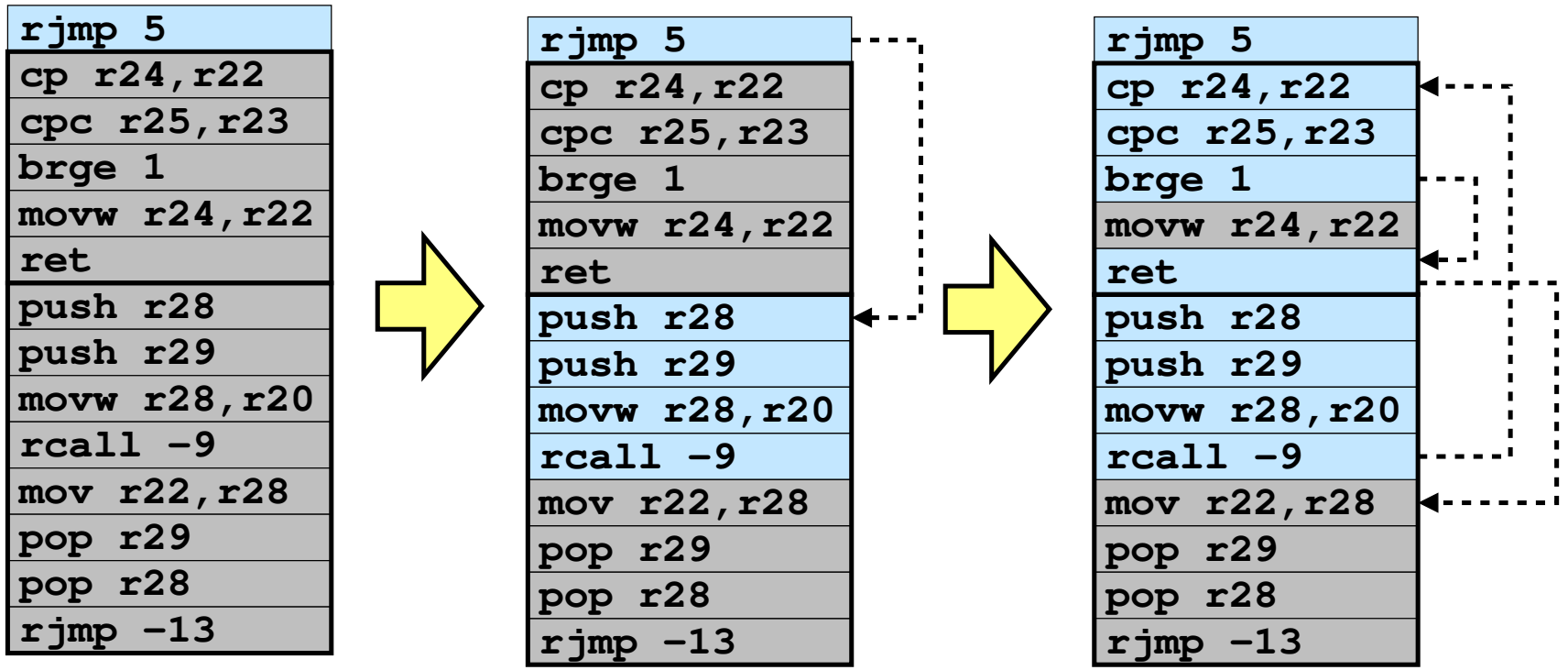
Example: `max3(int,int,int)`

```
int max(int x, int y) {
  if(x > y) {
    return x;
  } else {
    return y;
  }
}


int max3(int x,int y,int z){
  int xy = max(x,y);
  return max(xy,z);
}
```

```
0x00: rjmp 5
0x01: cp r24,r22 // max
0x02: cpc r25,r23
0x03: brge 1
0x04: movw r24,r22
0x05: ret
0x06: push r28  // max3
0x07: push r29
0x08: movw r28,r20
0x09: rcall -9
0x0A: movw r22,r28
0x0B: pop r29
0x0C: pop r28
0x0D: rjmp -13
```

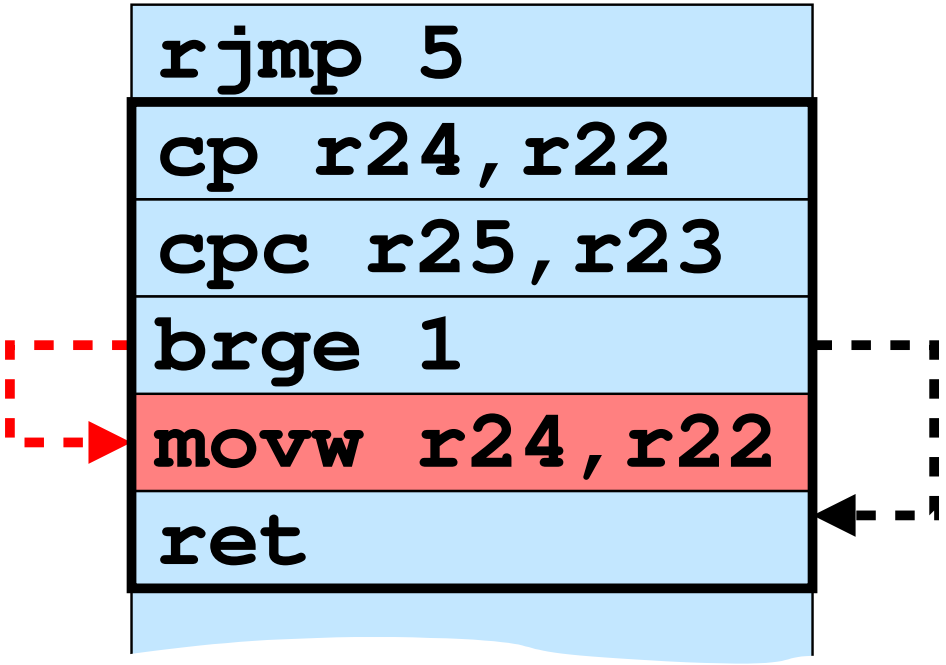
How to follow control-flow here?

Example: `max3(int,int,int)`



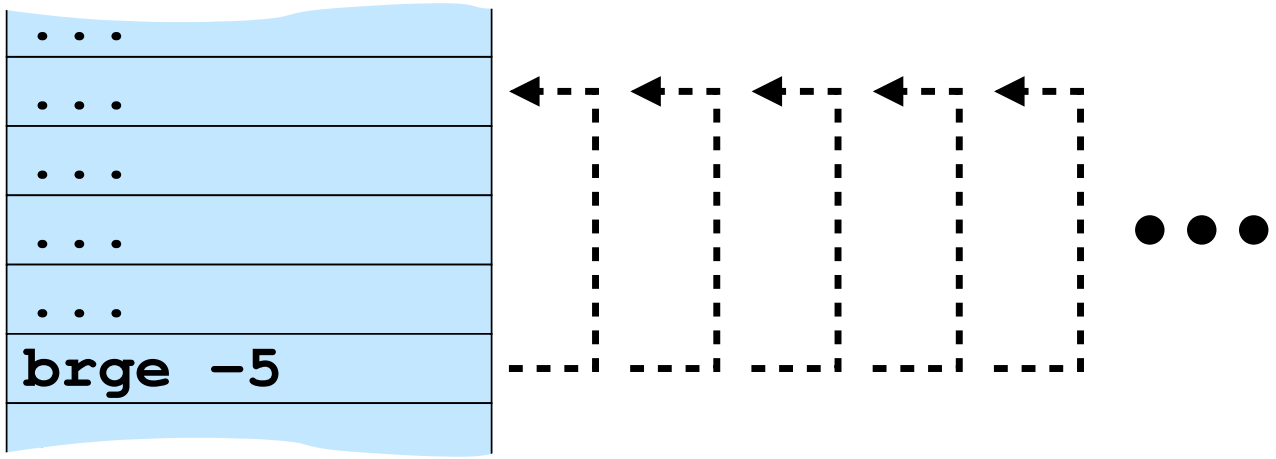
- Can **trace** execution flow from root (i.e. address `0x0000`).
- Works well for **direct branching** instructions (e.g. `rjmp`,`rcall`).
- But, what about **conditional branches?**

Example: `max3(int,int,int)`



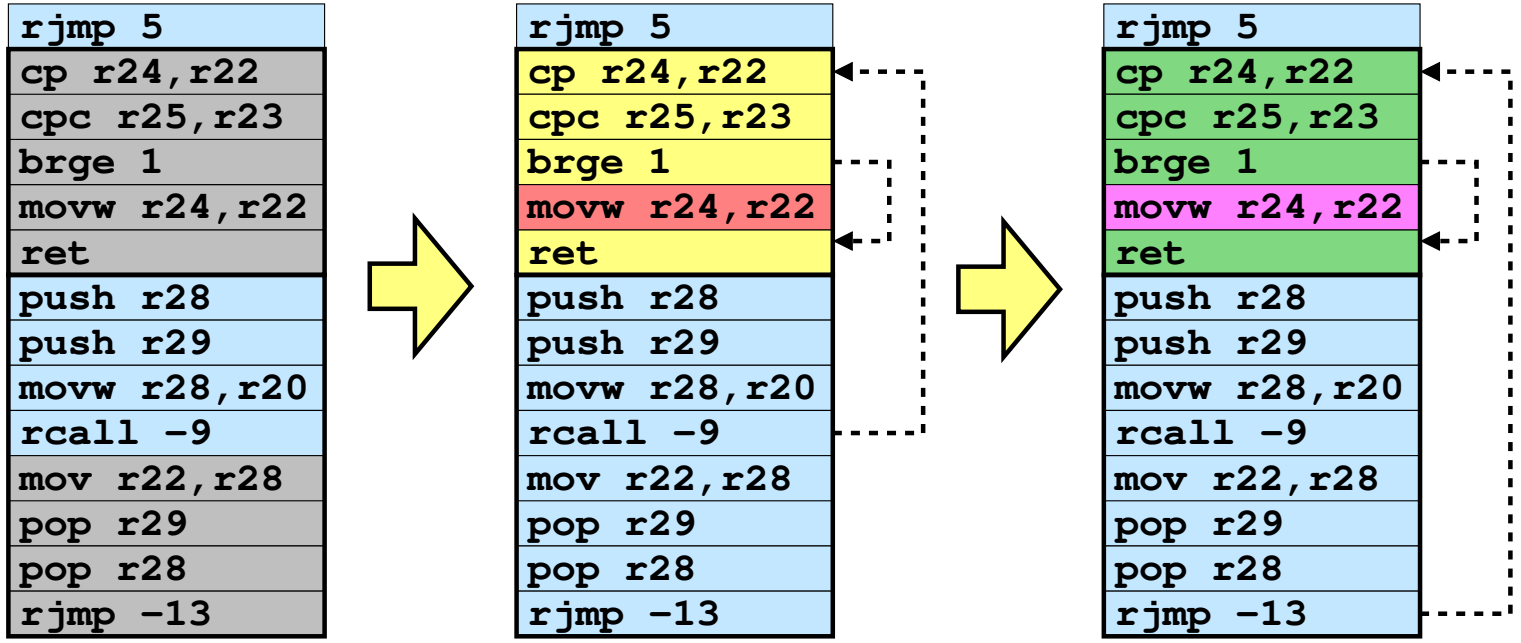
- Static analysis is at **compile time**!
- Therefore, cannot tell whether branch **taken** or **not**.
- Therefore, assume **both outcomes** possible!

Control-Flow Analysis: Loops



- What to do about **loops?**
- Simple traversal will just **loop forever**...
- Therefore, **terminate traversal** if instruction seen before.

Control-Flow Analysis: Method Calls



- But **method invocations** visit instructions more than once!
- Therefore, `call` instruction begins **new traversal**.
- And, `ret` instructions **terminates traversal**

Control-Flow Analysis: Indirect Branching

`IJMP` **— Indirect Jump**

Description: *Indirect jump to the address pointed to by the Z (16 bits) Pointer Register in the Register File. The Z- pointer Register is 16 bits wide and allows jump within the lowest 64K words (128KB) section of Program memory.*

- Why is this instruction such a **big problem?**
- How does this instruction **arise**?

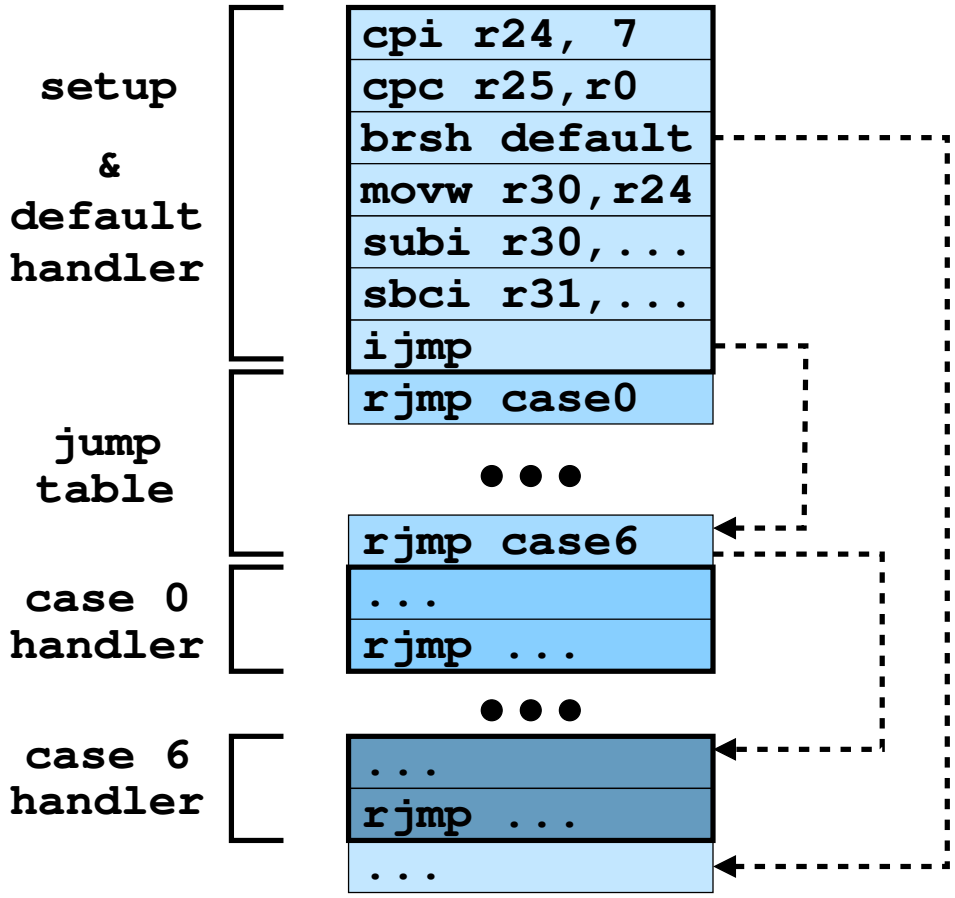
Jump Tables!

"In computer programming, a branch table or **jump table** is a method of transferring program control (branching) to another part of a program (or a different program that may have been dynamically loaded) using a table of branch or jump instructions. It is a form of multiway branch. The branch table construction is commonly used when programming in assembly language but may also be generated by a compiler, especially when implementing an **optimized switch statement** where known, small ranges are involved with few gaps."

–Wikipedia

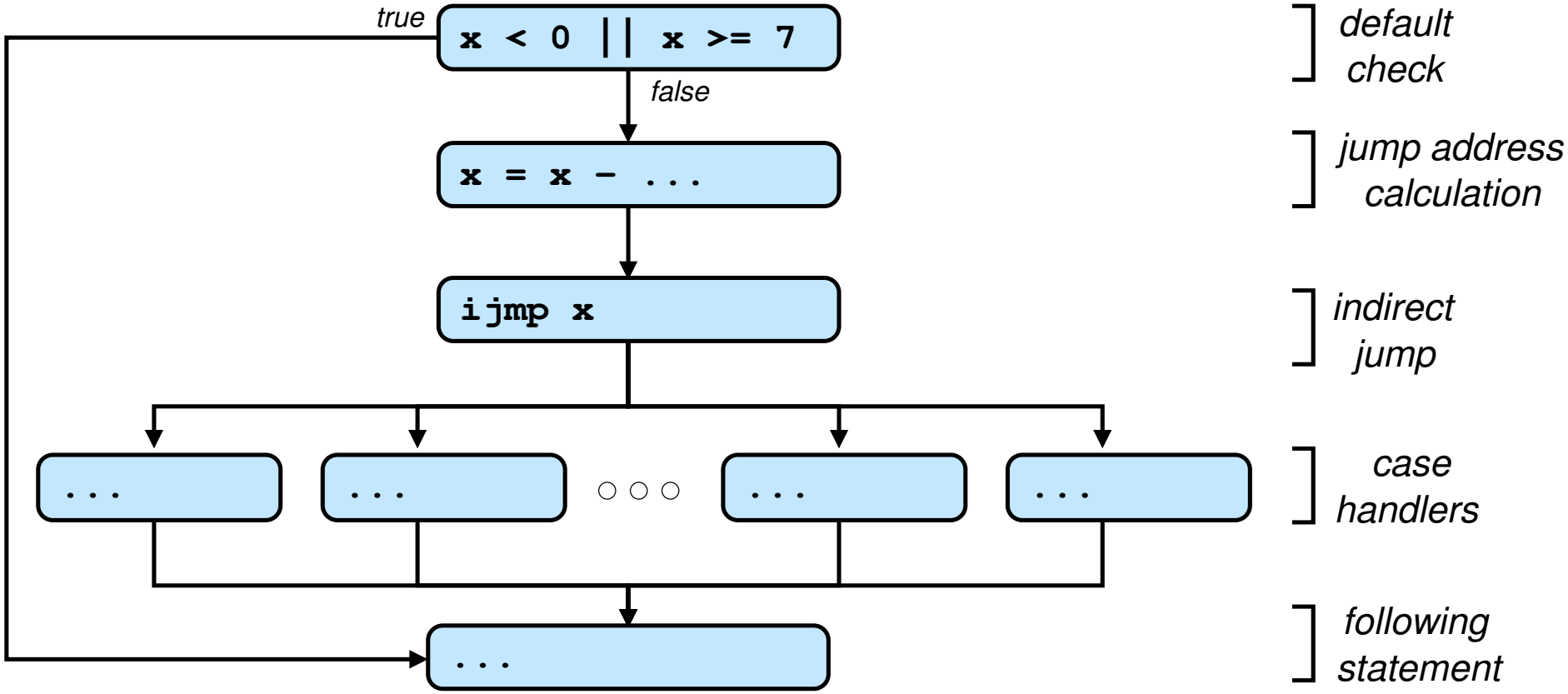
Control-Flow Analysis: Switch Statements

```
switch(x) {
  case 0:

    ...

  case 6:
    ...
  }
```



- Translation of `switch` statment via **jumptable** using `ijmp`
- Illustrating **execution path** for `default` case and "`case 6`"

Control-Flow Analysis: Switch Statements



- High level overview of `switch` translation

Integer Range Analysis: Overview

In computing, in particular compiler construction, value range analysis is a type of data flow analysis that tracks the **range (interval) of values** *that a numeric variable can take on at each point of a program's execution.*

–Wikipedia

Integer Range Analysis: Operations

Definition

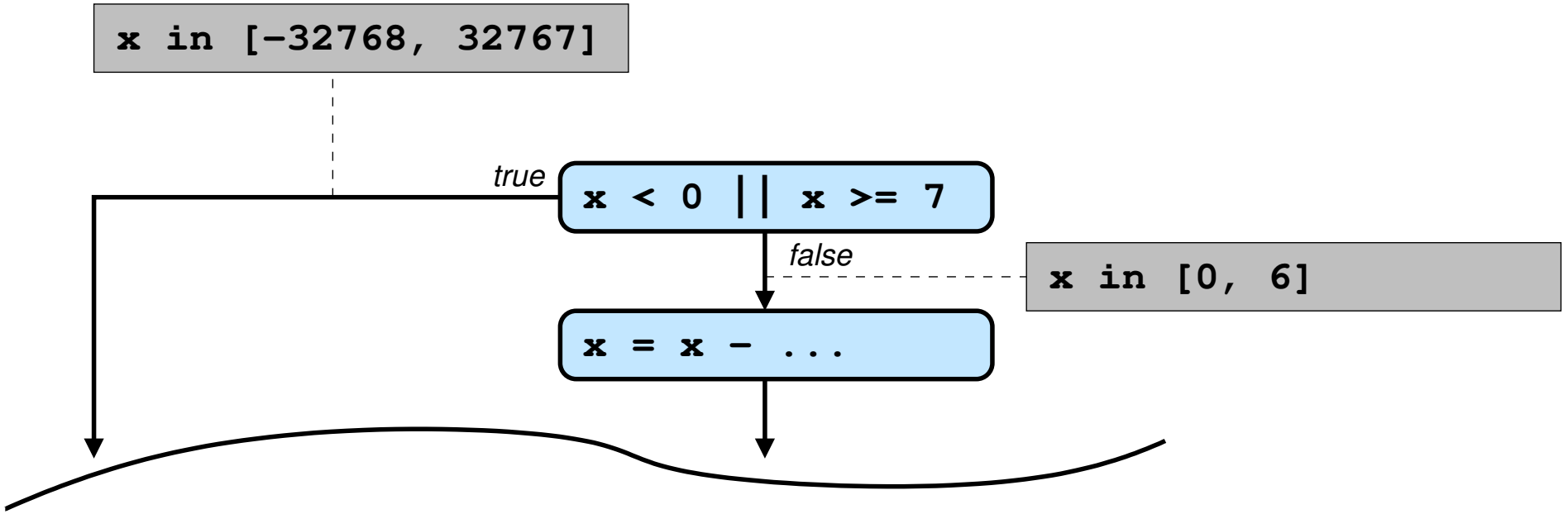
Let $[l, u]$ denote a range of integer values where l and u are either integer constants or $\pm\infty$, and where $[l, u] = \{x \mid x \in \mathbb{Z} \wedge l \leq x \wedge x \leq u\}$.

$$
\begin{array}{rcl}
[l_1, u_2] \; + \; [l_2, u_2] & \doteq & [l_1 + l_2, u_1 + u_2] \\
[l_1, u_2] \; - \; [l_2, u_2] & \doteq & [l_1 - u_2, u_1 - l_2] \\
[l_1, u_2] \; \times \; [l_2, u_2] & \doteq & [\min(xs), \max(xs)]
\end{array}
$$

$$\text{where } xs = \{l_1 \times l_2, l_1 \times u_2, u_1 \times l_2, u_1 \times u_2\}$$

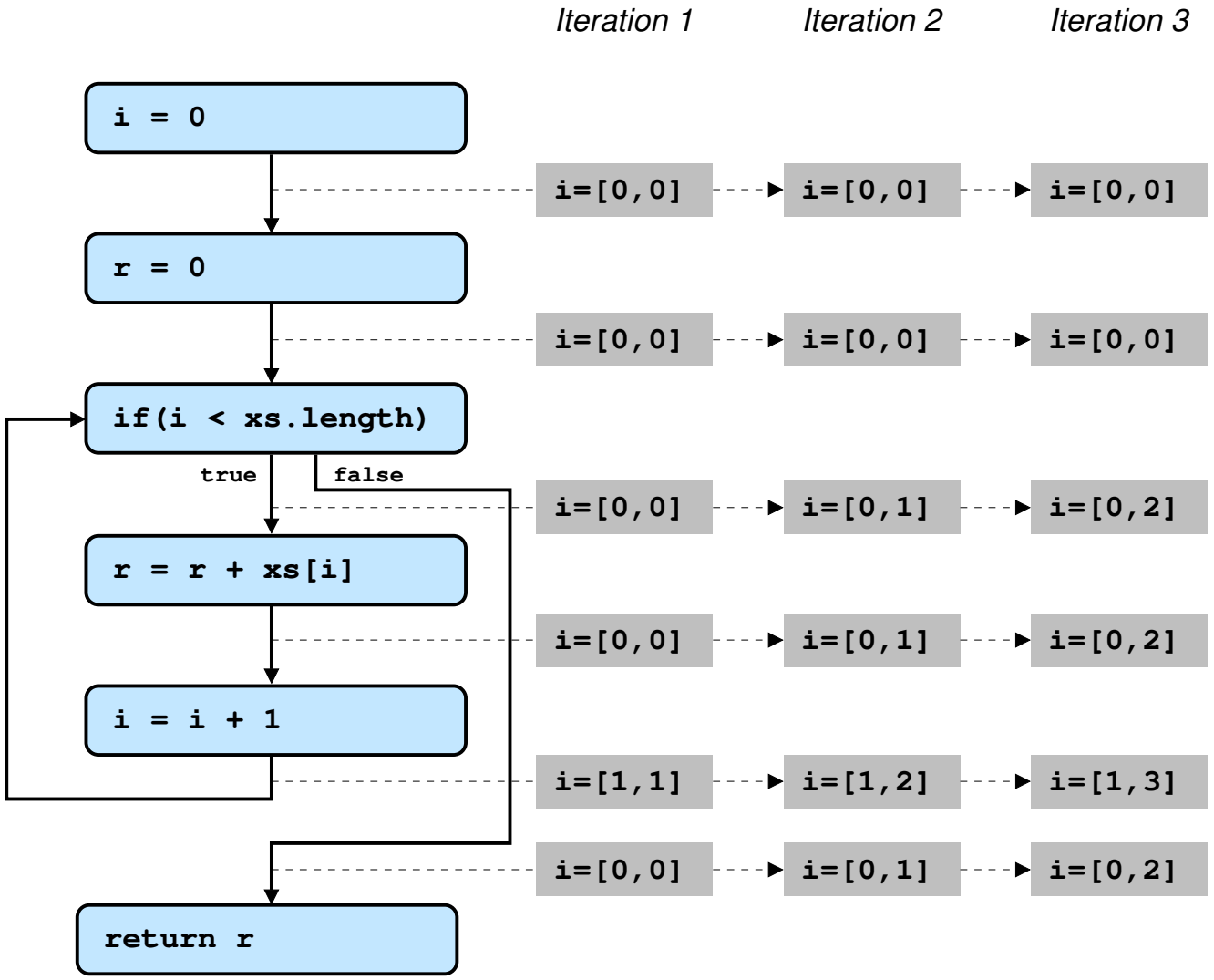
- **Example ($+$):** $[1, 6] \; + \; [2, 3] \Longrightarrow [3, 9]$
- **Example ($-$):** $[-\infty, 2] \; - \; [1, 2] \Longrightarrow [-\infty, 1]$
- **Example ($\times$):** $[2, 3] \; \times \; [1, 4] \Longrightarrow [2, 12]$

Integer Range Analysis: Switch Statements



- Indirect branch targets can now be **determined**!

Integer Range Analysis: Loops



Q) *When does it end?*

Further Reading

- **Eliminating stack overflow by abstract interpretation**. John Regehr, Alastair Reid and Kirk Webb. In *Transactions on Embedded Computing Systems*, 2005.

- **Proving the Absence of Stack Overflows**. Daniel Kästner and Christian Ferdinand. In *Proc. SAFECOMP*, 2014.

- **Integer Range Analysis for Whiley on Embedded Systems**. David J. Pearce. In *Proc. Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, pages 26–33, 2015.

- **The modulo interval: A simple and practical representation for program analysis**. T. Nakanishi, K. Joe, C. D. Polychronopoulos, and A. Fukuda. In *Proc. Pact*. 1999

- **Interval analysis and machine arithmetic: Why signedness ignorance is bliss**, G. Gange, J. A. Navas, P. Schachte, H. Sondergaard, and P. J. Stuckey. In *TOPLAS*, 2015.