Verification of Real-Time Systems

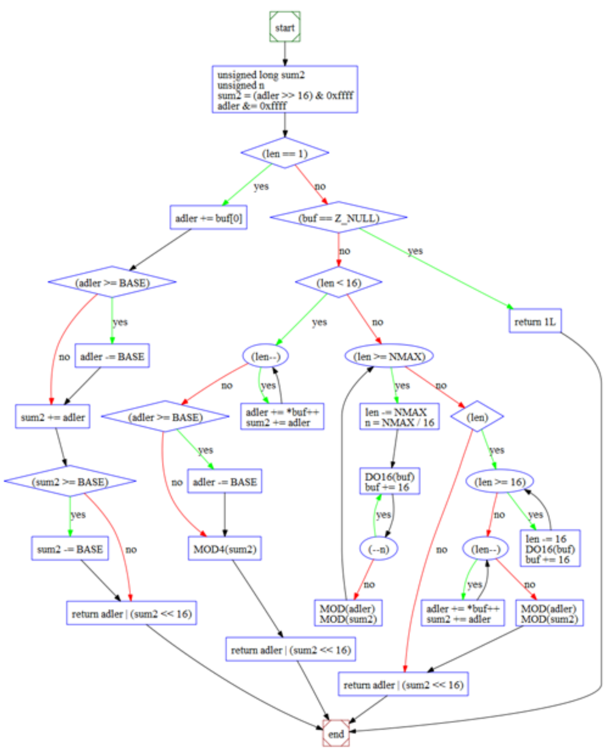
Static WCET Analysis

Jan Reineke

Advanced Lecture, Summer 2015

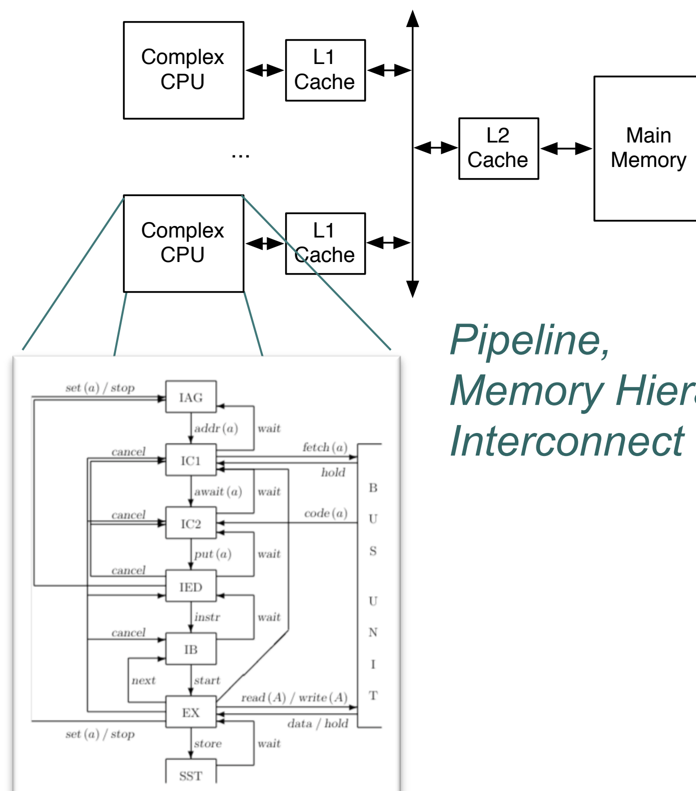
What does the execution time of a program depend on?

Input-dependent control flow



+

Microarchitectural State



Pipeline,
Memory Hierarchy,
Interconnect

Formalization of WCET Analysis Problem

Consider all possible program inputs

Consider all possible initial states of the hardware

$$WCET_H(P) := \max_{i \in Inputs} \max_{h \in States(H)} ET_H(P, i, h)$$

Measuring the execution time for all inputs and all hardware states is not feasible in practice:

- *There are too many.*
- *We cannot control the initial hardware states.*

➔ *Need for approximation!*

High-level Requirements for WCET Analysis

- Upper bounds must be safe, i.e. not underestimated.
- Upper bounds should be tight, i.e. not far away from real execution times.
- Analysis effort must be tolerable.

Standard WCET Analysis Approach Today: Divide and Conquer + Abstraction

1. Divide: split program into fragments (e.g. basic blocks).

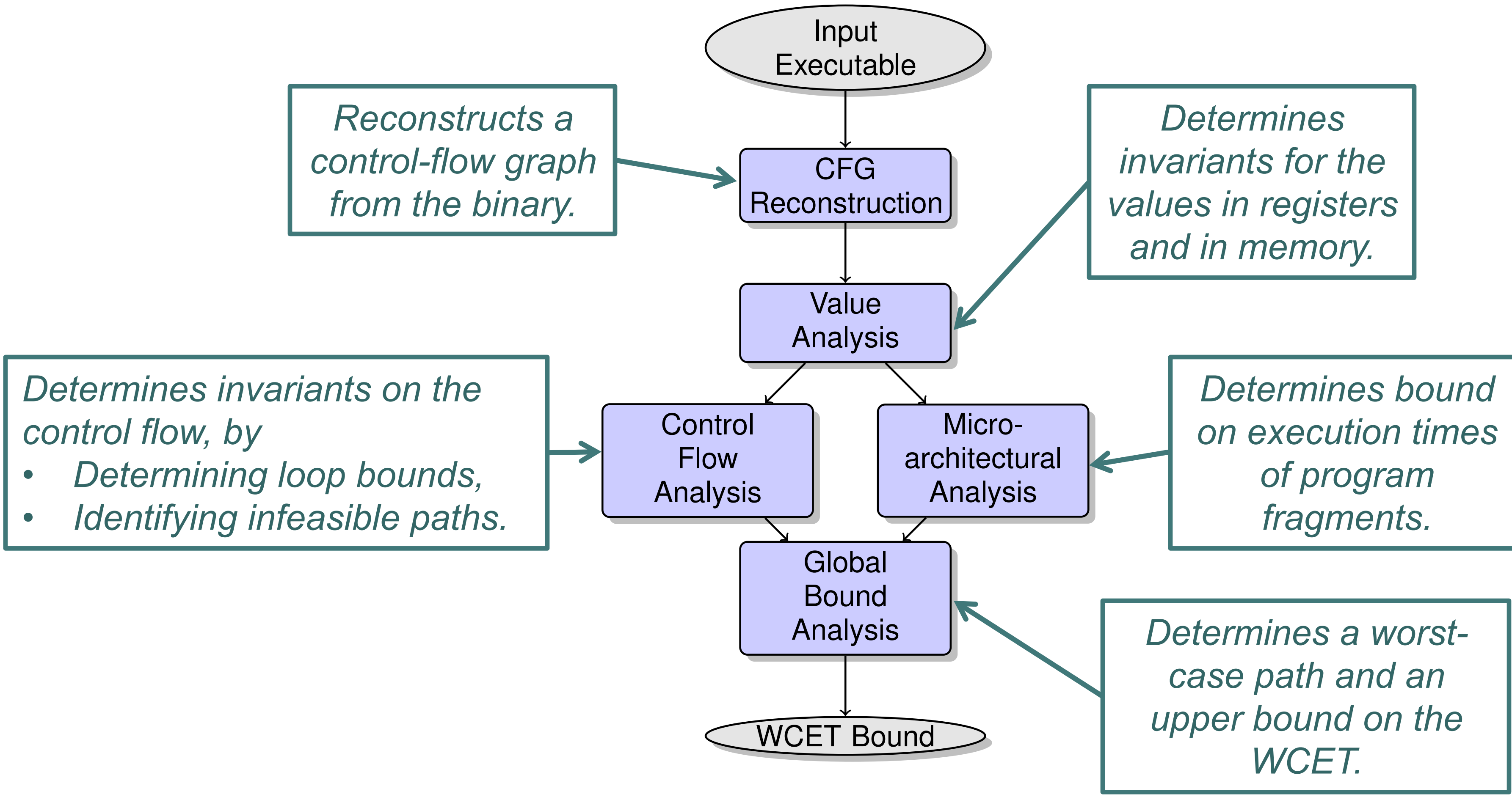
HW - DEPENDENT

2. Determine safe bounds on execution time of each fragment using abstractions.

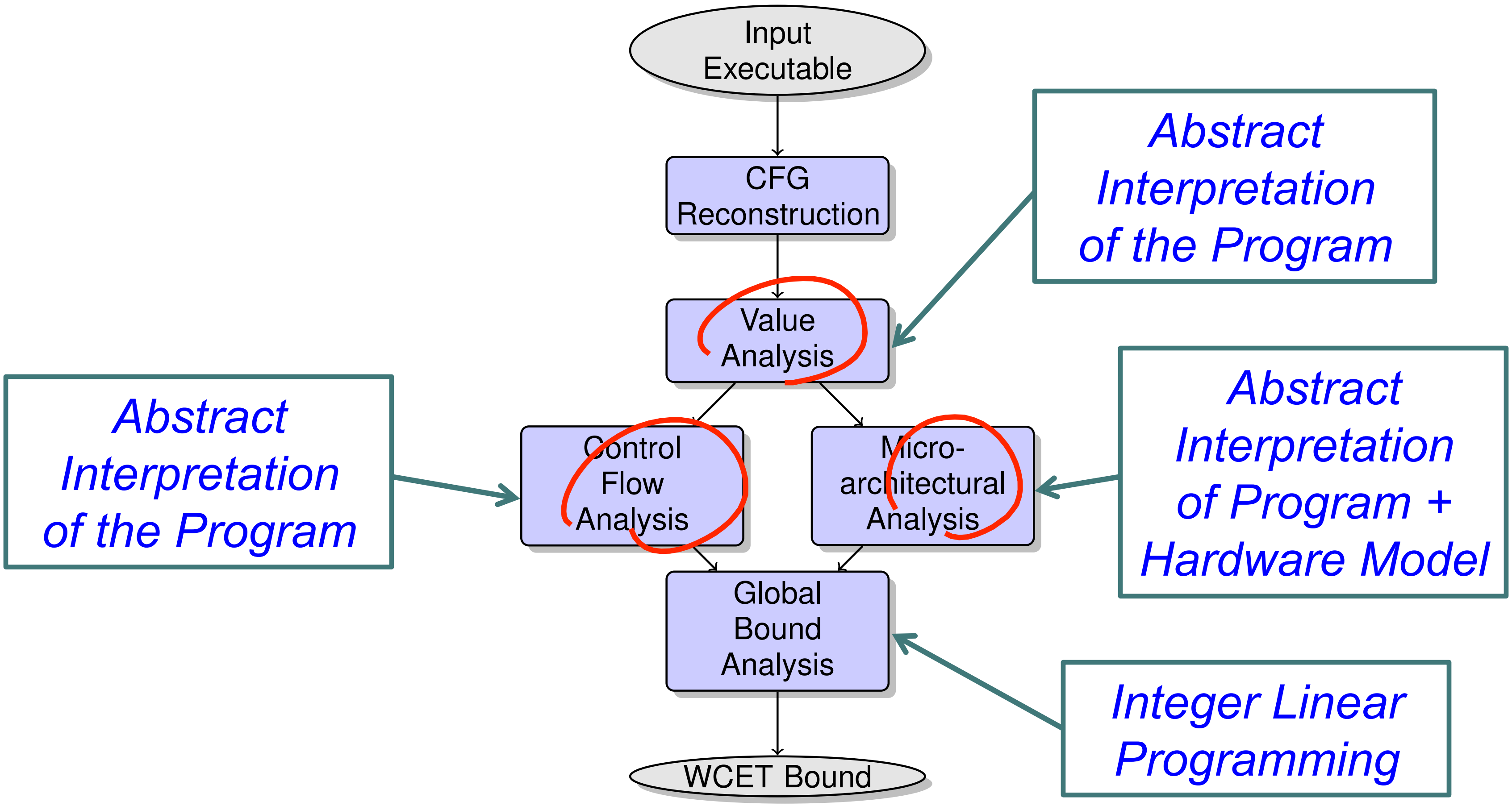
SW - DEPENDENT

3. Determine constraints on control flow (e.g. loop bounds) through program by abstractions.
4. Conquer: combine 2 + 3 into bound of execution time of the whole program.

Structure of WCET Analyzers

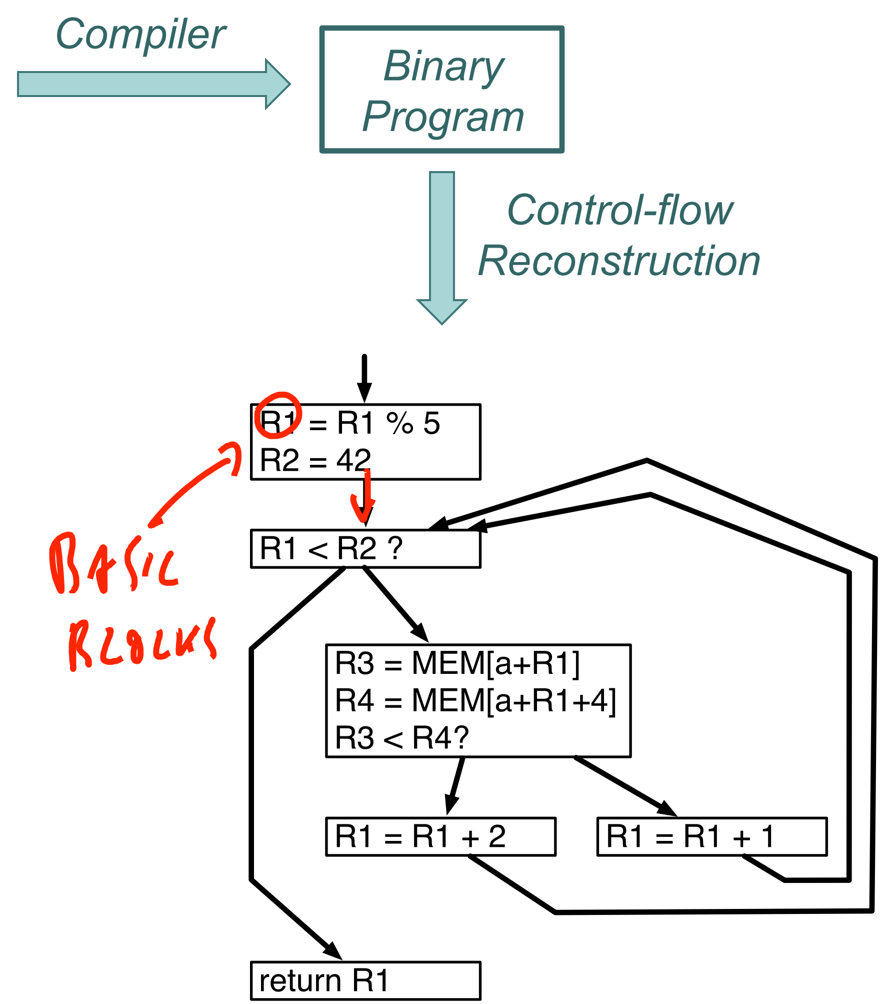


Structure of WCET Analyzers
Employed Techniques

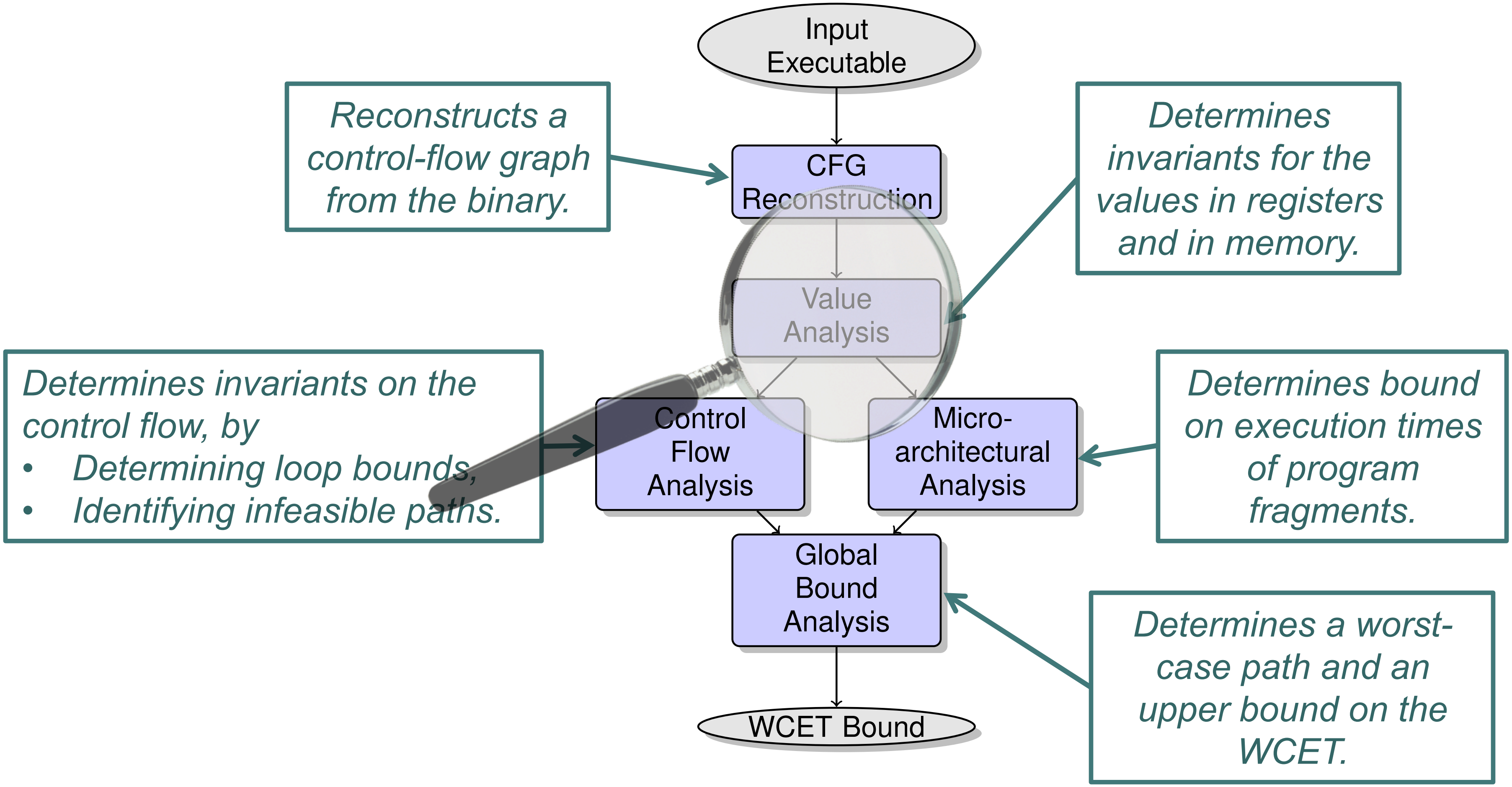


Running Example

```
int main(int x, int[] a) {
    int x = x % 5;
    int y = 42;
    while (x < y) {
        if (a[x] < a[x+1])
            x++
        else
            x += 2;
    }
    return x;
}
```



Structure of WCET Analyzers

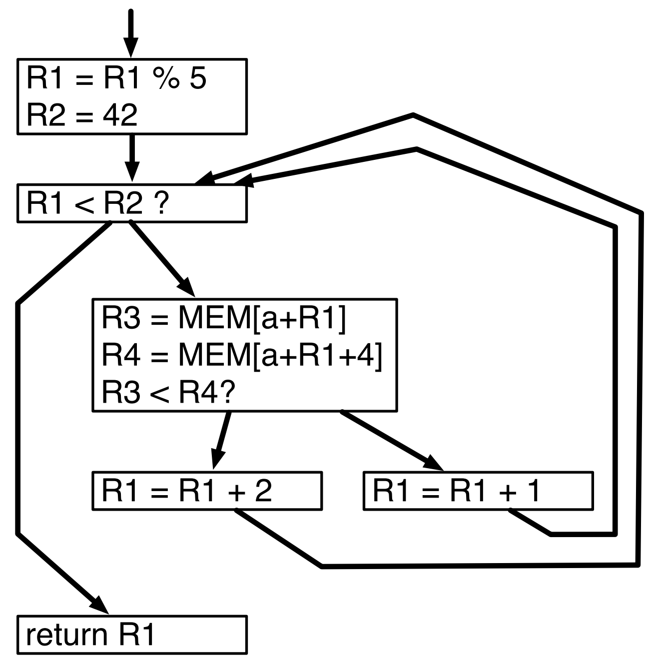


Value Analysis

Determines invariants on values of registers at different program points. Invariants are often in the form of enclosing intervals of all possible values.

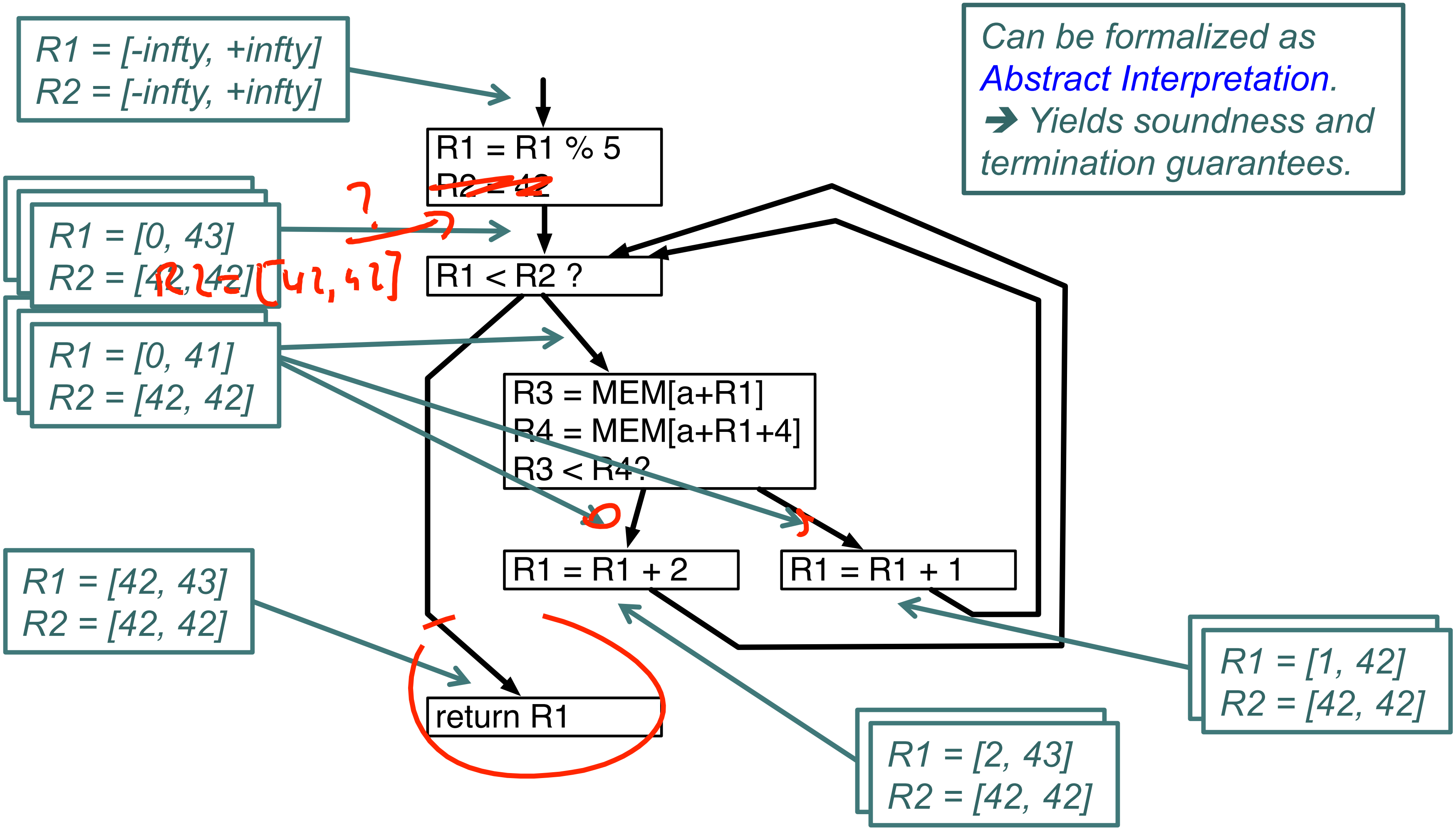
Where is this information used?

- Microarchitectural Analysis
  - Pipeline Analysis
  - Cache Analysis
- Control-Flow Analysis
  - Detect infeasible paths
  - Derive loop bounds



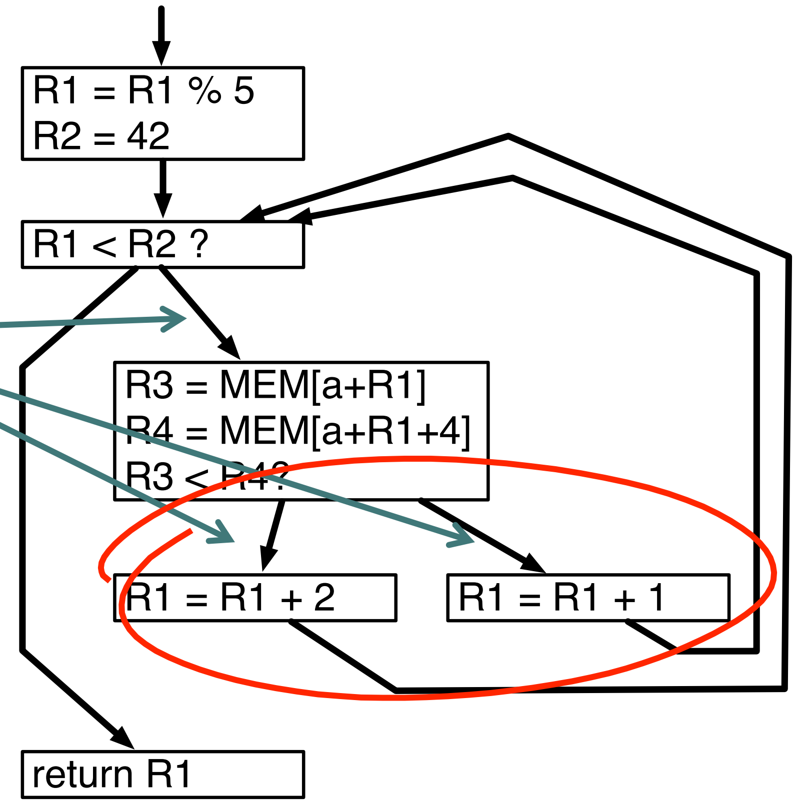
Value Analysis
Intuition of Interval Analysis

Can be formalized as [*Abstract Interpretation*.]
➔ Yields soundness and termination guarantees.



Control-Flow Analysis

R1 = [0, 41]
R2 = [42, 42]



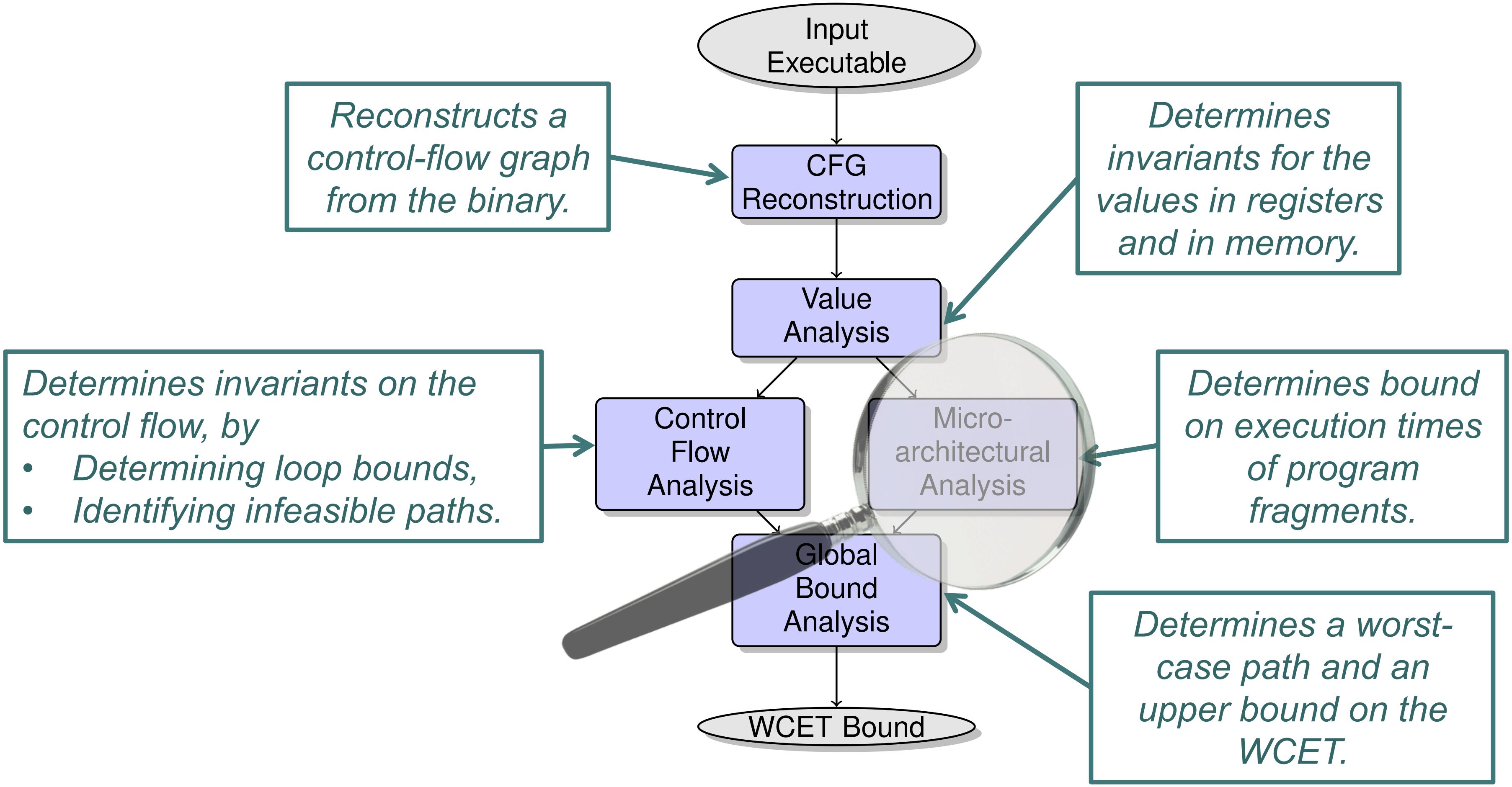
R1 increases by at least 1 in every iteration

➔ *Can enter loop at most 42 times*

$\frac{4n-4}{2} \pm 1$

Can we also come up with a lower bound?

Structure of WCET Analyzers



Microarchitectural Analysis

Ideal 1970s world: one instruction = one cycle

Real world:

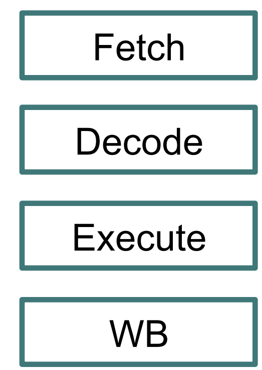
- Pipelining
- Branch prediction + speculative execution
- Caches
- DRAM

→ Execution time of individual instruction highly variable and dependent on state of microarchitecture

→ Need to determine in which states the microarchitecture may be at a point in the program

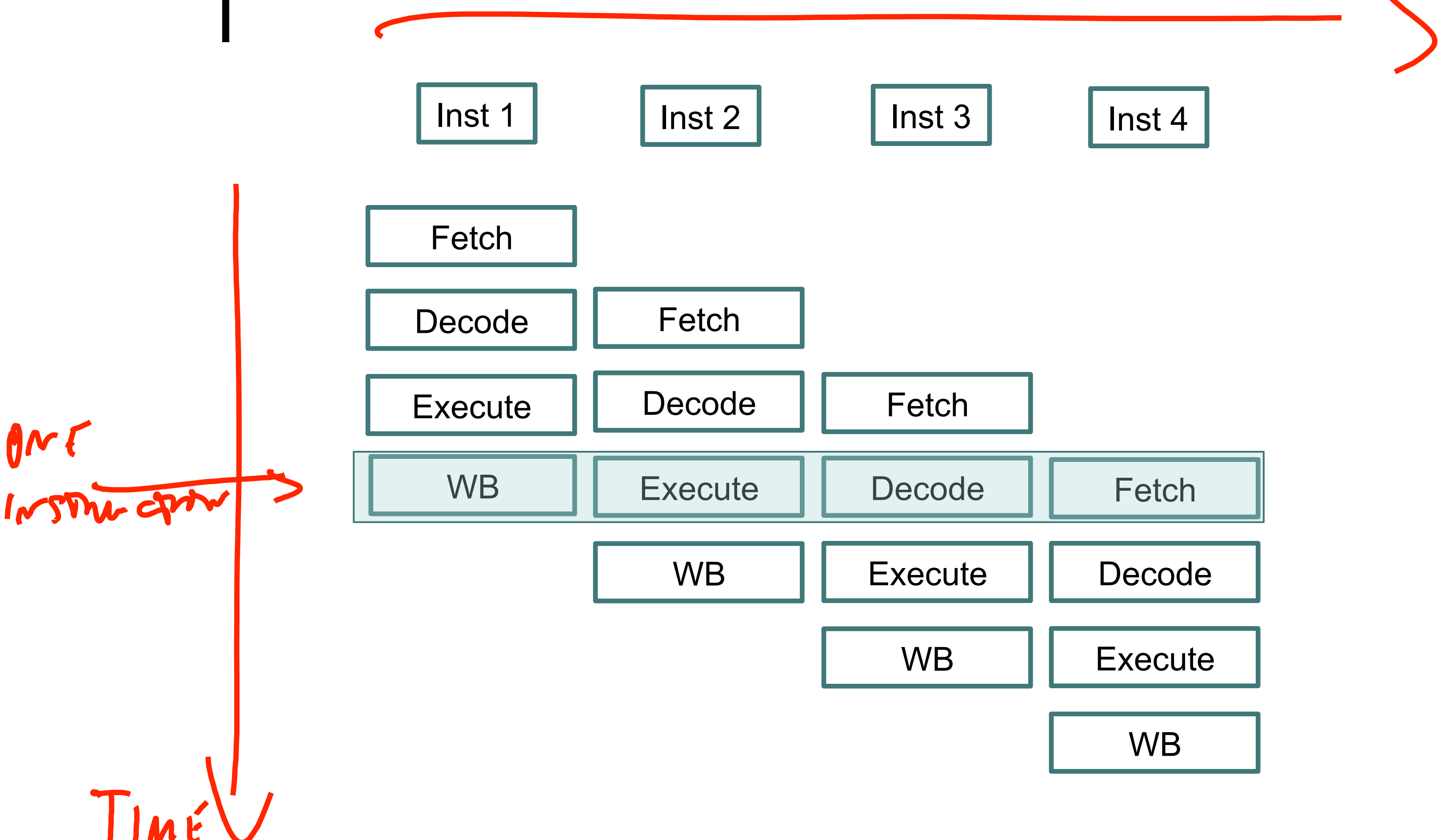
Pipelining

- Instruction execution is split into several stages
- Several instructions can be executed in parallel
- Some pipelines can start more than one instruction per cycle: VLIW, Superscalar
- Some processors can execute instructions out-of-order
- Practical Problems: Hazards and cache misses



Hardware Features: Pipelines

program



Ideal Case: One Instruction per Cycle

Pipeline Hazards

Pipeline Hazards:

- Data Hazards: Operands not yet available (Data Dependences)
- Resource Hazards: Consecutive instructions use same resource
- Control Hazards: Conditional branch
- Instruction-Cache Hazards: Instruction fetch causes cache miss
- DATA - CACHE

Assuming worst case everywhere is not an option!

Static exclusion of hazards

Cache analysis: prediction of cache hits on instruction or operand fetch or store

lwz r4, 20(r1)

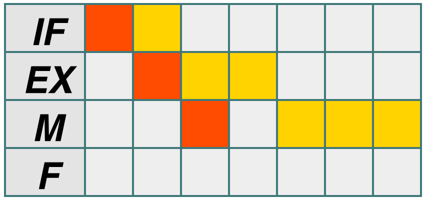
Hit

Dependence analysis: elimination of data hazards

add r4, r5,r6
lwz r7, 10(r1)
add r8, r4, r4

Operand ready

Resource reservation tables: elimination of resource hazards



View of Processor as a State Machine

- Processor (pipeline, cache, memory, inputs) viewed as a *big* state machine, performing transitions every clock cycle
- Starting in an initial state for an instruction, transitions are performed, until a final state is reached:
  - End state: instruction has left the pipeline
  - # transitions: execution time of instruction

A Concrete Pipeline Executing a Basic Block

function exec (*b* : **basic block**, *s* : **concrete pipeline state**)
t: **trace**

interprets instruction stream of *b* starting in state *s* producing trace *t.*

Successor basic block is interpreted starting in initial state *last(t)*

length(t) gives number of cycles for basic block *b*

An Abstract Pipeline Executing a Basic Block

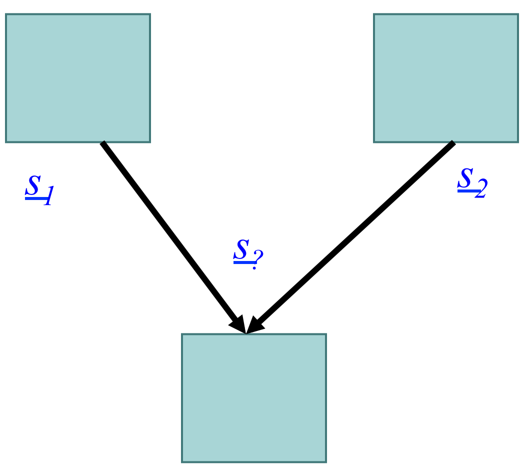
function exec (*b* : **basic block**, *s* : **abstract pipeline state**) *t*: **trace**

interprets instruction stream of *b* (annotated with cache information) starting in state *s* producing abstract trace *t*

length(t) gives number of cycles

What is different?

- Abstract states may lack information, e.g. about cache contents.
- More than one trace may be possible.
- Starting state for successor basic block?
  In particular, if there are several predecessor blocks.



Alternatives:

- *sets of states*
- *combine by least upper bound (join), hard to find one that*
  - *preserves information and*
  - *has a compact representation.*

Nondeterminism

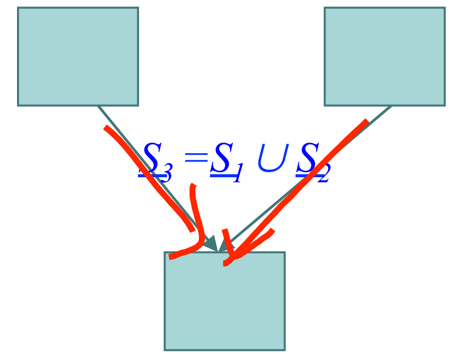
- In the concrete pipeline model, one state resulted in one new state after a one-cycle transition
- Now, in the abstract model, one state can have several successor states
  - Transitions from set of states to set of states

Non-Locality of Local Contributions

- Interference between processor components produces **Timing Anomalies**:
  - Assuming local best case leads to higher overall execution time.
  - Assuming local worst case leads to shorter overall execution time
    Ex.: Cache miss in the context of branch prediction
- Treating components in isolation may be unsafe
- Implicit assumptions are not always correct:
  - Cache miss is not always the worst case!
  - The empty cache is not always the worst-case start!

An Abstract Pipeline Executing a Basic Block

function analyze (*b* : **basic block**, $\underline{S}$: **analysis state**) $\underline{T}$: **set of trace**



Analysis states = $2^{\underline{PS} \times \underline{CS}}$

PS = set of abstract pipeline states

CS = set of abstract cache states

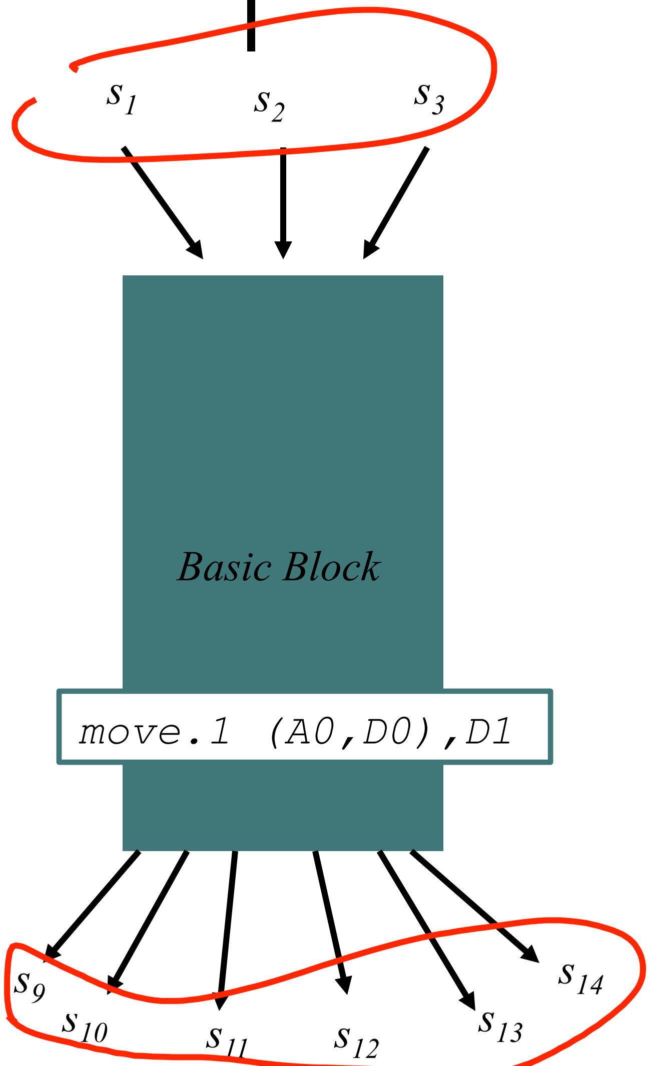
interprets instruction stream of *b* (annotated with cache information) starting in state $\underline{S}$ producing set of traces $\underline{T}$

max(length($\underline{T}$*))* - upper bound for execution time

last($\underline{T}$*)* - set of initial states for successor block

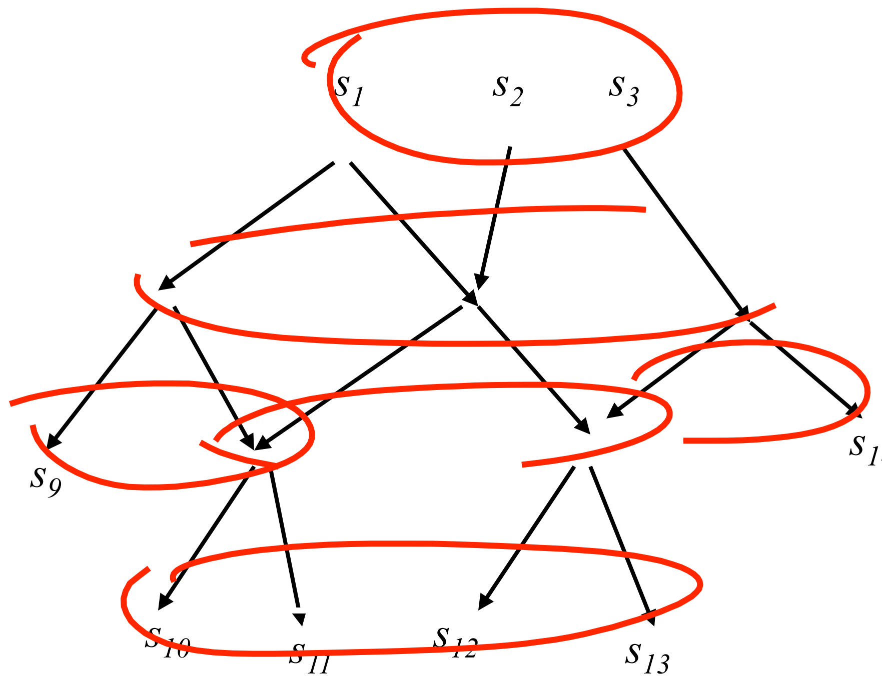
Union for blocks with several predecessors.

Integrated Analysis: Overall Picture

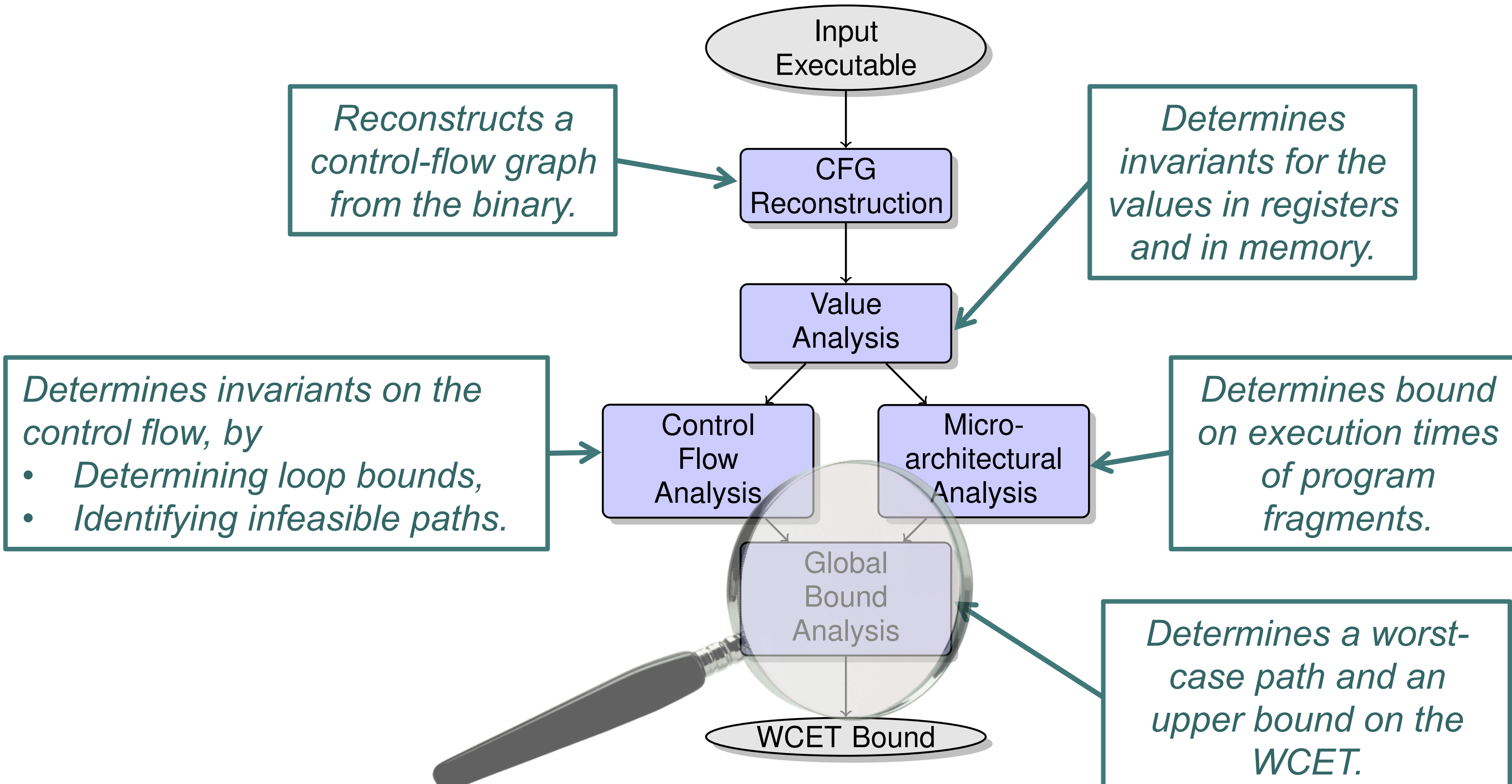


Fixed point iteration over Basic Blocks in abstract state $\{s_1, s_2, s_3\}$

Cyclewise evolution of processor model for instruction

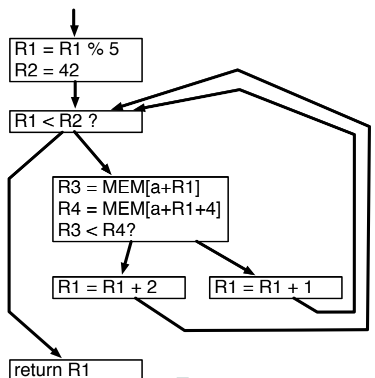


Structure of WCET Analyzers



Global Bound Analysis
aka Path Analysis aka Implicit Path Enumeration

- Determines a worst-case path and an upper bound on the WCET.
- Formulated as integer linear program (ILP).



+

Loop bounds + Infeasible paths

+

Execution time bounds for basic blocks

Integer
Linear
Program

Integer linear programming

Linear programming (LP)

Objective function → $$maximize \;\; c^T x$$

Linear constraints → $$subject\ to \;\; Ax \leq b$$

$$and \;\; x \geq 0$$

… + Restriction to integers = ILP.

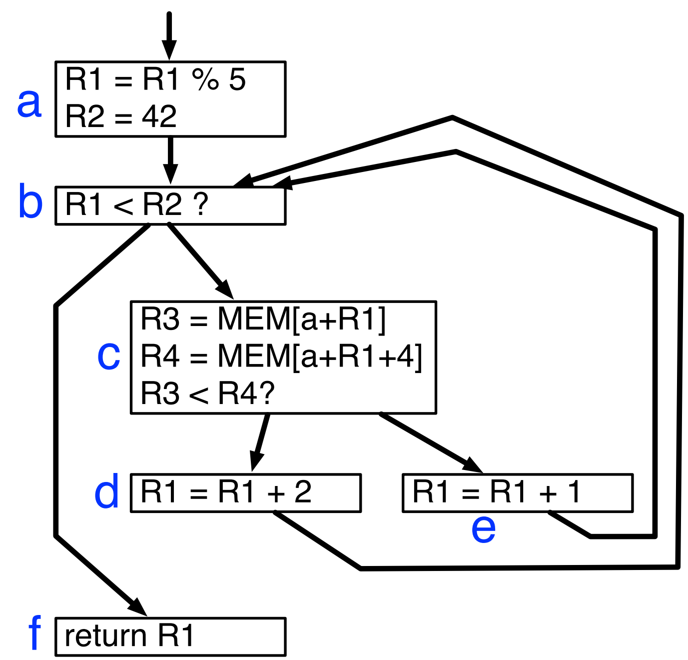
LP is in polynomial time, yet, ILP is NP hard,

but often efficiently solvable in practice.

Solvers (e.g. CPLEX) determine the maximal value of the objective function + corresponding valuation of variables.

Global Bound Analysis
aka Path Analysis aka Implicit Path Enumeration

- Determines a worst-case path and an upper bound on the WCET.
- Formulated as integer linear program (ILP).



x_b = frequency of executing b
c_b = time to execute b once

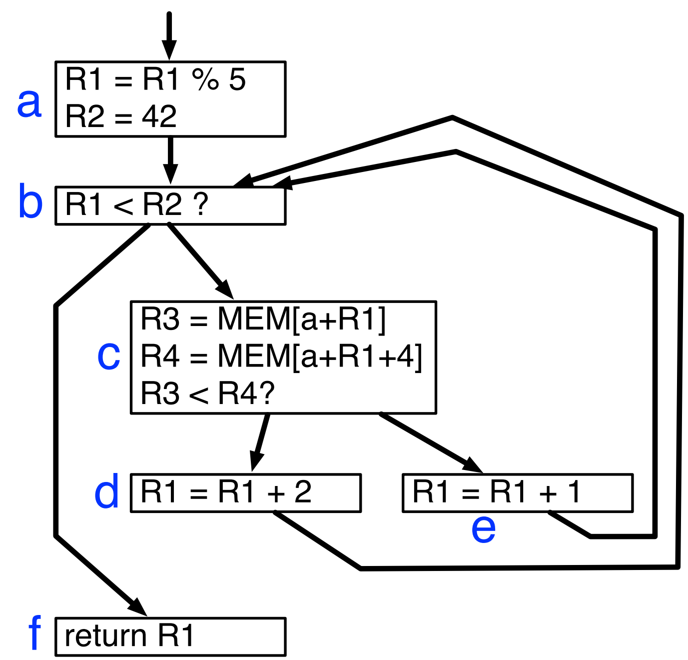
$$\begin{aligned}
\max\ & c_a x_a + c_b x_b + c_c x_c + \\
& c_d x_d + c_e x_e + c_f x_f \\
\text{s.t.}\ & x_b = x_a + x_d + x_e \\
& x_c = x_d + x_e \\
& x_a = x_f = 1 \\
& x_a >= 0,\ x_b >= 0,\ \ldots \\
& lb <= x_c <= ub
\end{aligned}$$

Structural constraints, due to CFG

Loop bounds

Global Bound Analysis
aka Path Analysis aka Implicit Path Enumeration

- Determines a worst-case path and an upper bound on the WCET.
- Formulated as integer linear program (ILP).



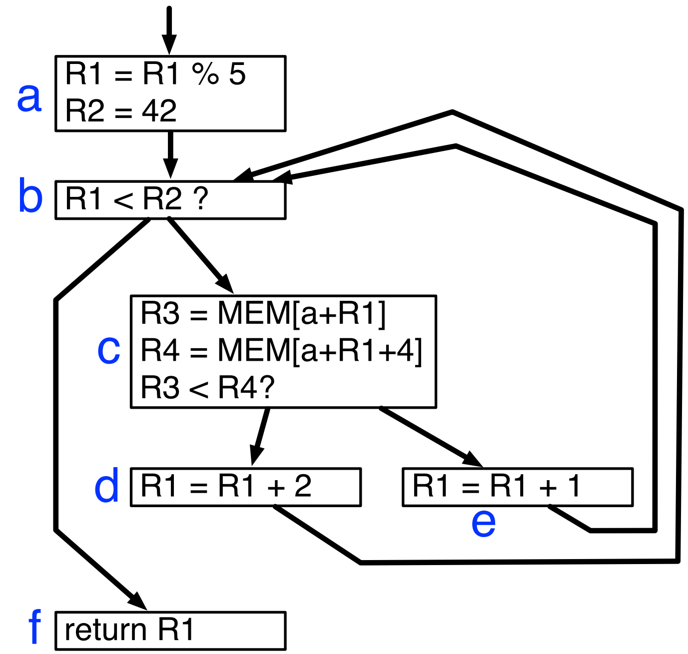
x_b = frequency of executing b
c_b = time to execute b once

$$
\begin{aligned}
\max\ & 2x_a + 3x_b + 6x_c + \\
& 3x_d + 2x_e + 2x_f \\
\text{s.t.}\ & x_b = x_a + x_d + x_e, \\
& x_c = x_d + x_e, \\
& x_a = x_f = 1, \\
& x_a >= 0,\ x_b >= 0,\ \ldots \\
& 19 <= x_c <= 42
\end{aligned}
$$

Structural constraints, due to CFG

Loop bounds

Global Bound Analysis
aka Path Analysis aka Implicit Path Enumeration



x_b = frequency of executing b
c_b = time to execute b once

$$\max\ 2x_a + 3x_b + 6x_c + 3x_d + 2x_e + 2x_f$$

$$\text{s.t.}\quad x_b = x_a + x_d + x_e,\quad x_c = x_d + x_e,\quad x_a = x_f = 1,\quad x_a >= 0,\ x_b >= 0,\ \ldots\quad 19 <= x_c <= 42$$

Structural constraints, due to CFG

Loop bounds

Solution:

$x_a = x_f = 1$, $x_b = 43$, $x_c = x_d = 42$

Objective function = 2*1 + 3*43 + (6+3)*42 + 2*1 = 511

Summary and Outlook

- Divide and conquer:
  - Analyze worst-case timing of program fragments separately
  - Combine results using integer linear program
- Abstraction:
  - Employ sound abstractions to solve undecidable problems approximately

Next week:

theoretical background of Abstract Interpretation