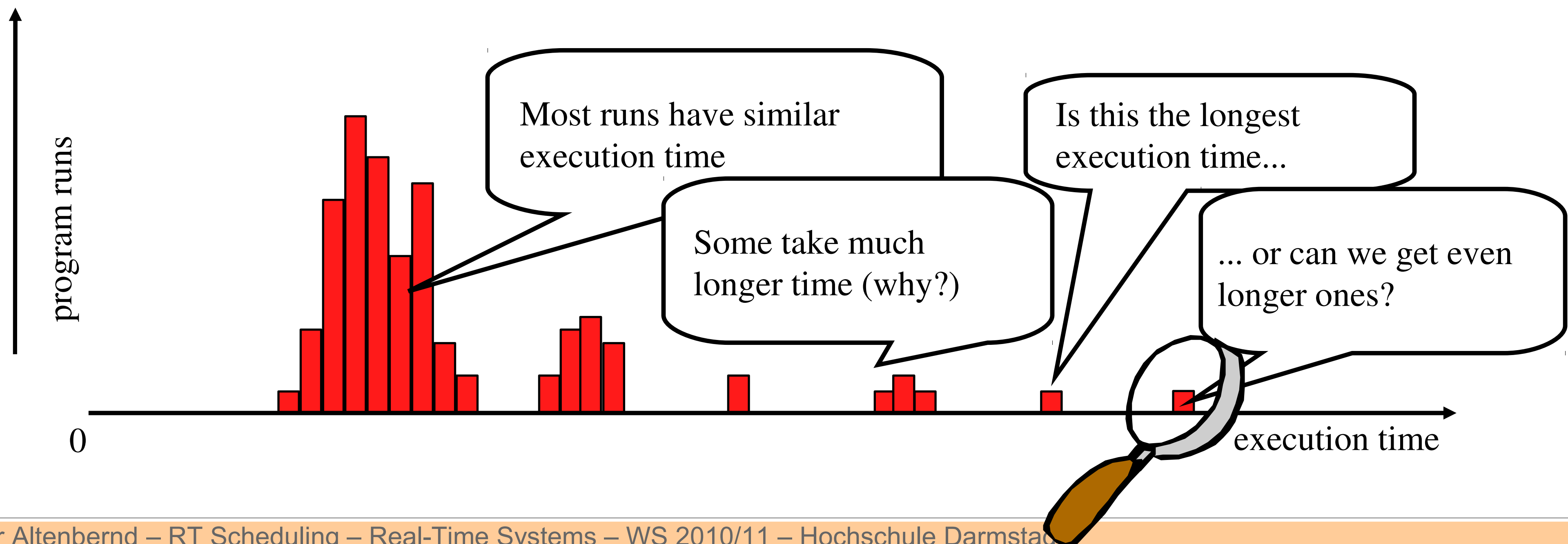
Vorlesung Real-Time Systems

2. Exkurs: WCET Analyse

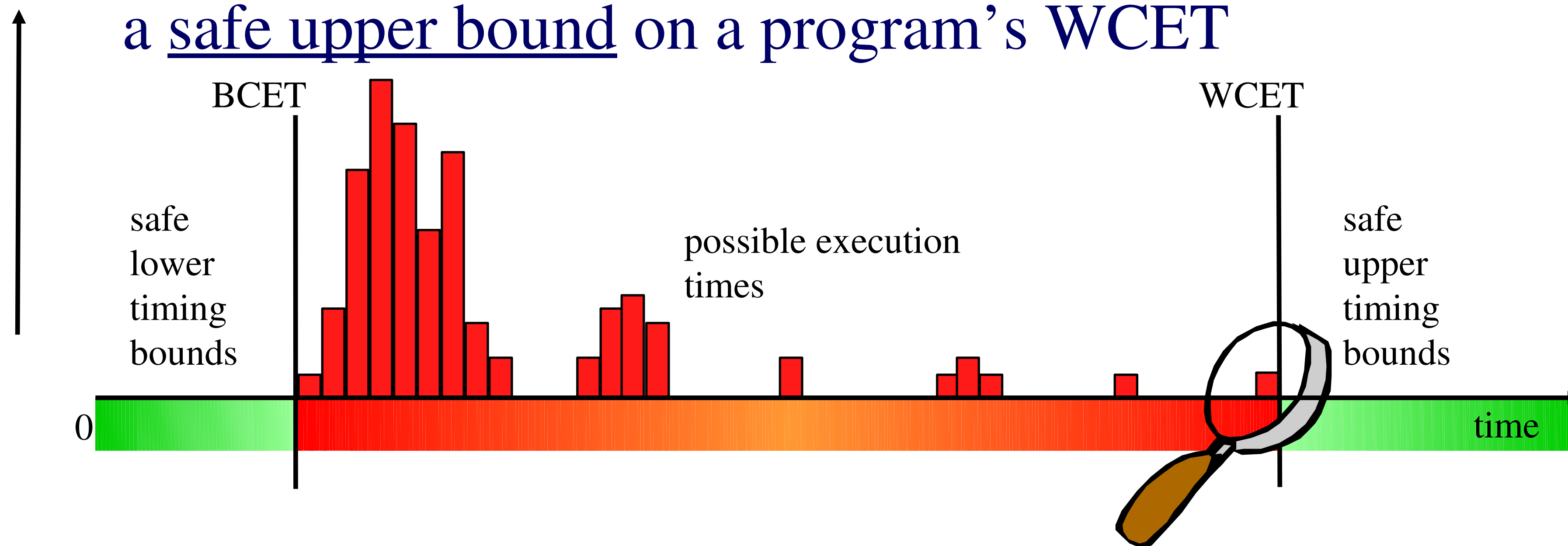
Program timing basics

- Most computer programs have varying execution time
  - Due to input values
  - Due to software characteristics
  - Due to hardware characteristics
- Example: some timed program runs



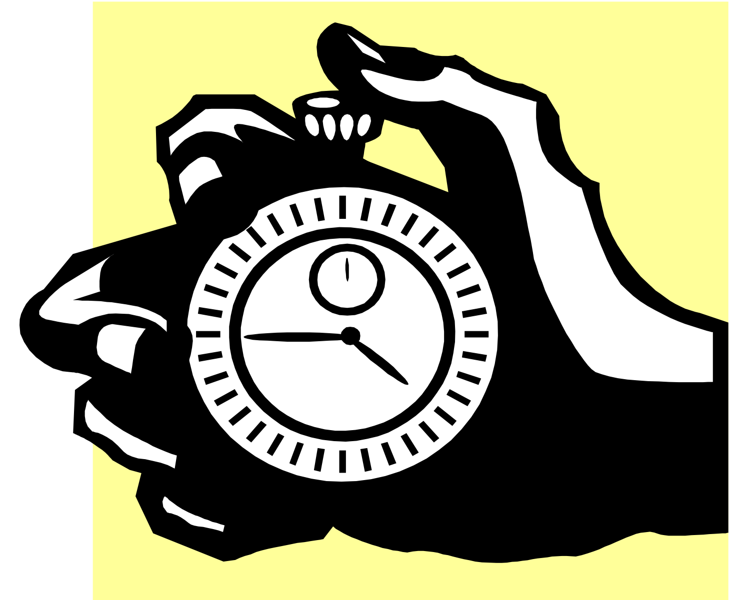
WCET and WCET analysis

- Worst-Case Execution Time = WCET
  - **The longest calculation time possible**
  - **For one program/task when run in isolation**
  - **Other interesting measures: BCET, ACET**
- The goal of a WCET analysis is to derive a <u>safe upper bound</u> on a program's WCET



Measuring for the WCET

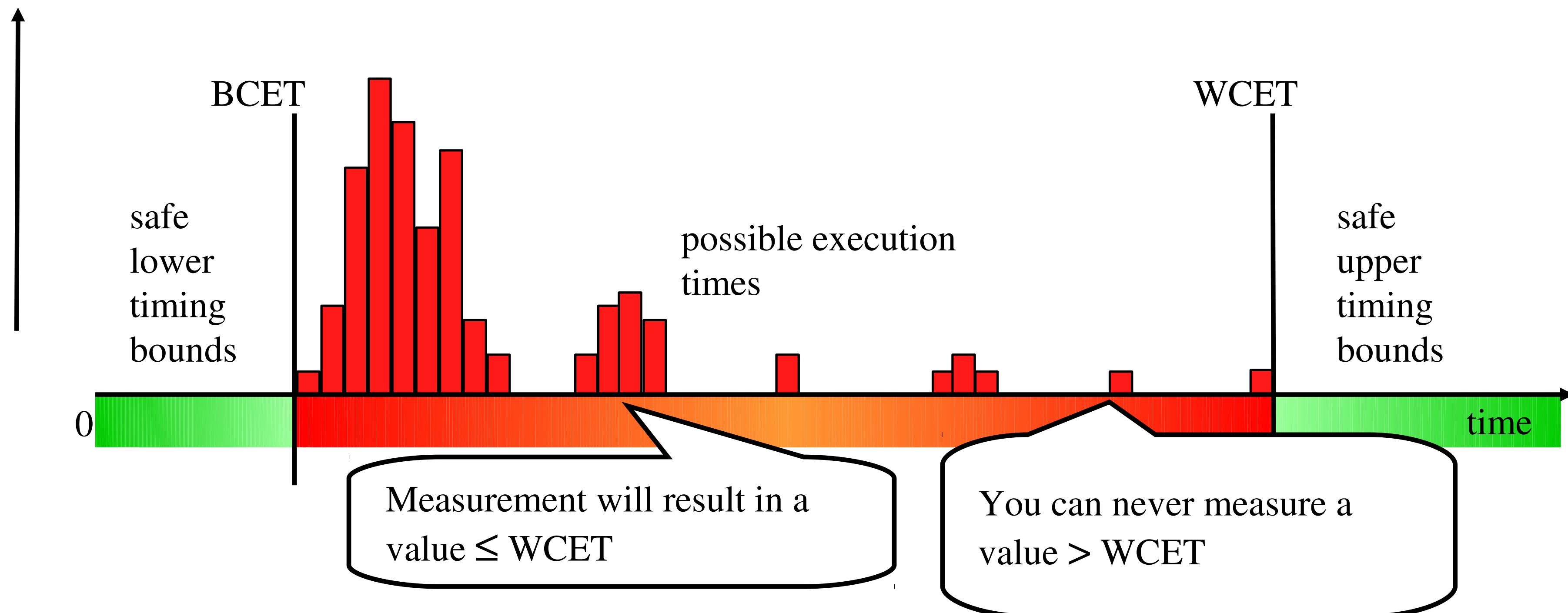
- Methodology:
  - Determine potential ”worst-case input”
  - Run and measure
  - Add a safety margin



- Large number of potential worst-case inputs
  - Program state might be part of input
- Has the worst-case path really been taken?
  - Often many possible paths through a program
  - Hardware features may interact in unexpected ways
- How to monitor the execution?
  - The instrumentation may affect the timing
  - How much instrumention output can be handled?

Problem of using measurement

- Measured value never larger than WCET!

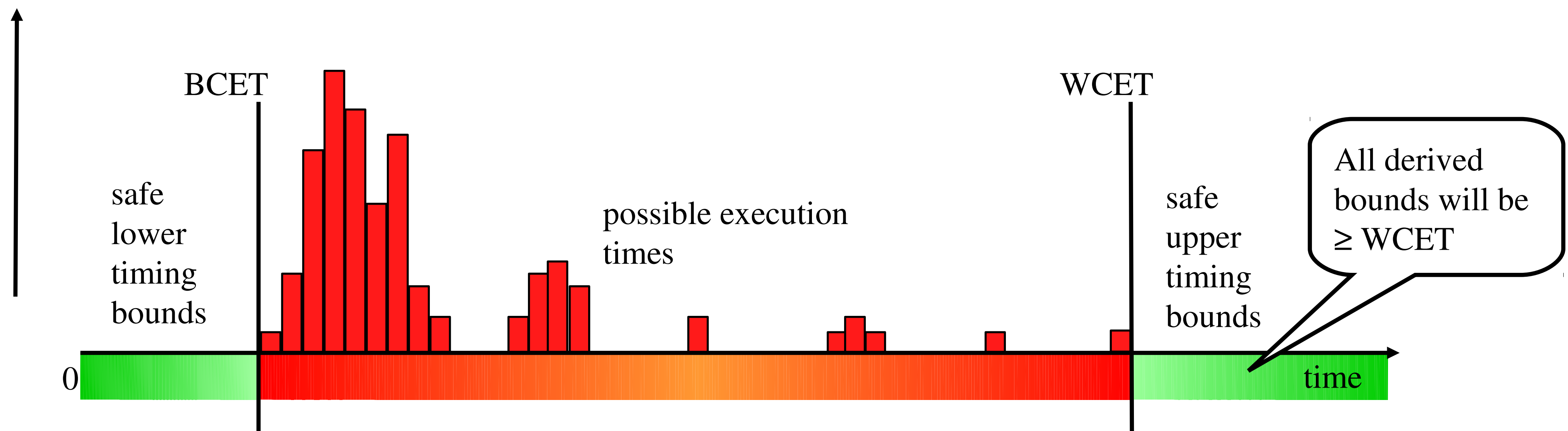


A safety margin must be added!

- How much is enough?

Static WCET analysis

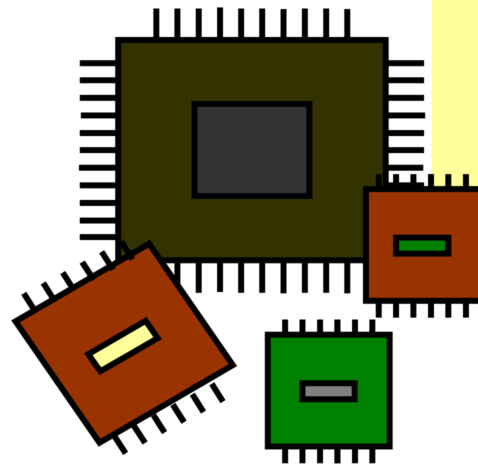
- Do not run the program – analyze it!
  - Using models based on the static properties of the software and the hardware
- Guaranteed reliable WCET bounds
  - Provided all models, input data and analysis methods are correct
- Trying to be as tight as possible



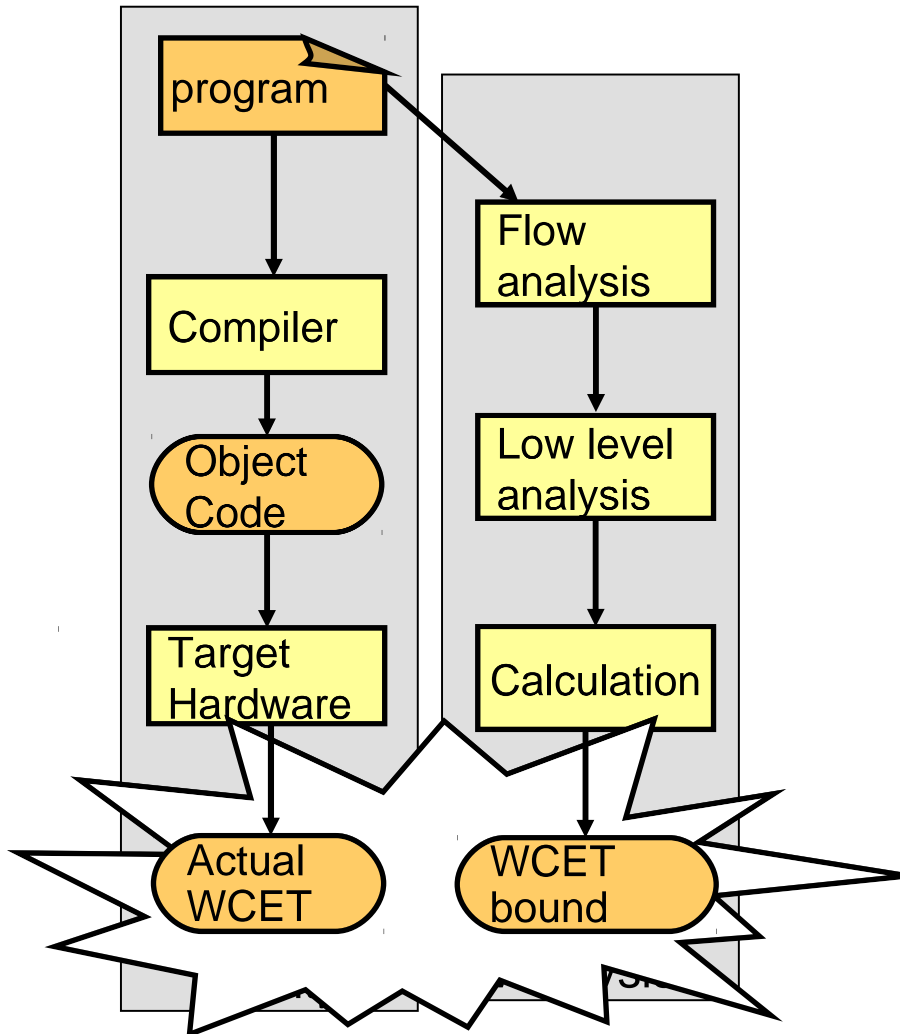
Again: Causes of Execution Time Variation

- Execution characteristics of the software
  - A program can often execute in many different ways
  - Input data dependencies
  - Application characteristics
- Timing characteristics of the hardware
  - Clock frequency
  - CPU characteristics
  - Memories used
  - …

```
foo(x,i):
  while(i < 100)
     if (x > 5) then
       x = x*2;
     else
       x = x+2;
     end
     if (x < 0) then
       b[i] = a[i];
     end
     i = i+1;
  end
```

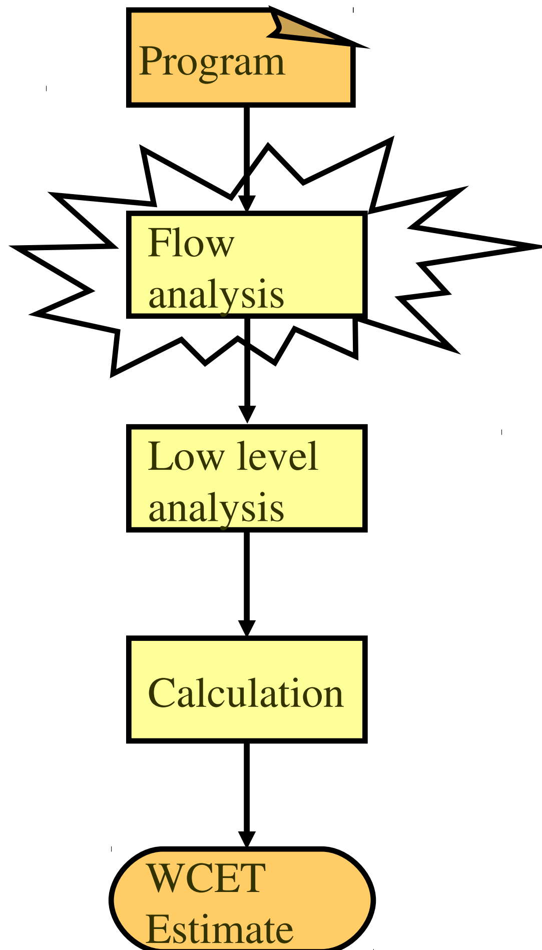


WCET analysis phases



- Flow analysis
  - Bound the number of times different program parts may be executed (SW analysis)
- Low-level analysis
  - Bound the execution time of different program parts (HW analysis)
- Calculation
  - Combine flow- and low-level analysis results to derive an upper WCET bound

Flow Analysis

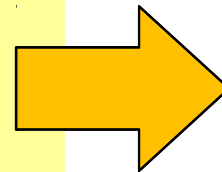


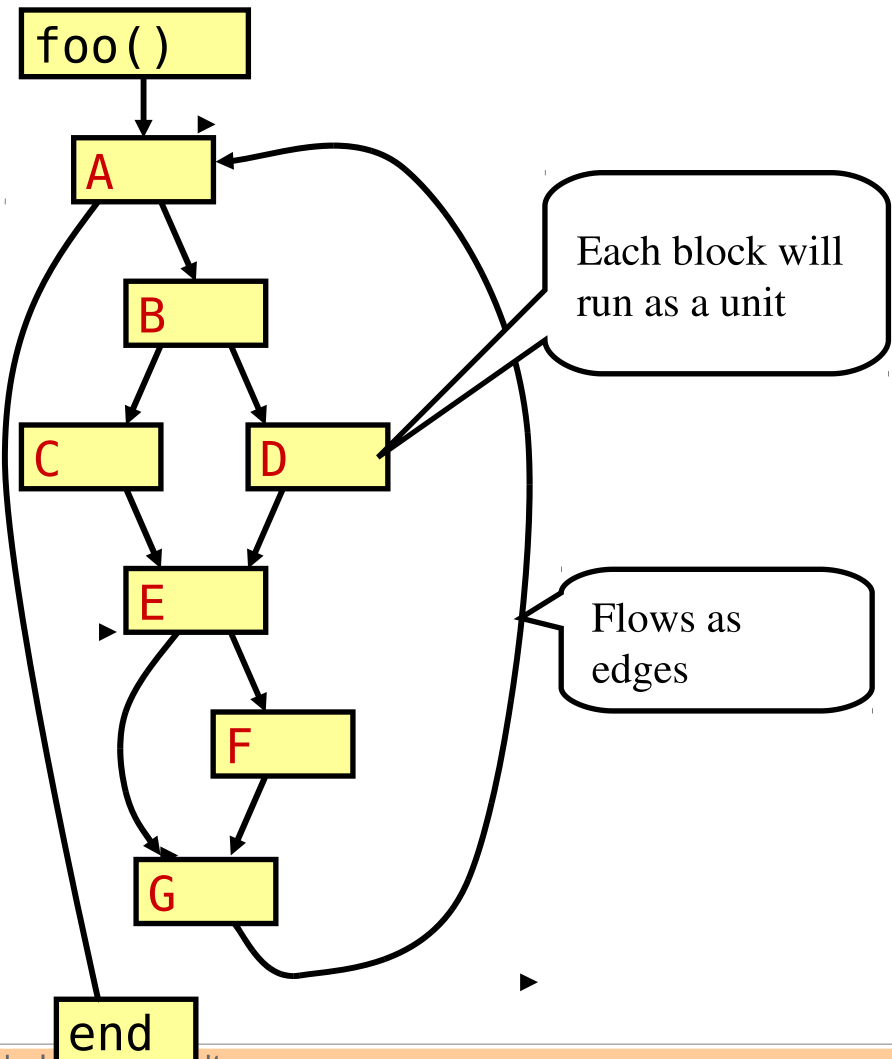
- Provides bounds on the number of times different program parts may be executed
  - Valid for all possible executions
- Examples of provided info:
  - Bounds of loop iterations
  - Bounds on recursion depth
  - Infeasible paths
- Info provided by:
  - Static program analysis
  - Manual annotations

10

The control-flow graph

```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```





Example: Loop bounds

```
    foo(x,i):
A:   while(i < 100)
B:     if (x > 5) then
C:       x = x*2;
       else
D:       x = x+2;
       end
E:     if (x < 0) then
F:       b[i] = a[i];
       end
G:     i = i+1;
     end
```

- **Loop bound:**
  - Depends on possible values of input variable i
    - E.g. if $1 \leq i \leq 10$ holds for input value i then loop bound is 100
  - In general, a very difficult problem
  - However, solvable for many types of loops
- **Requirement for basic finiteness**
  - All loops must be upper bound

Example: Infeasible path

```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i=i+1;
      end
```

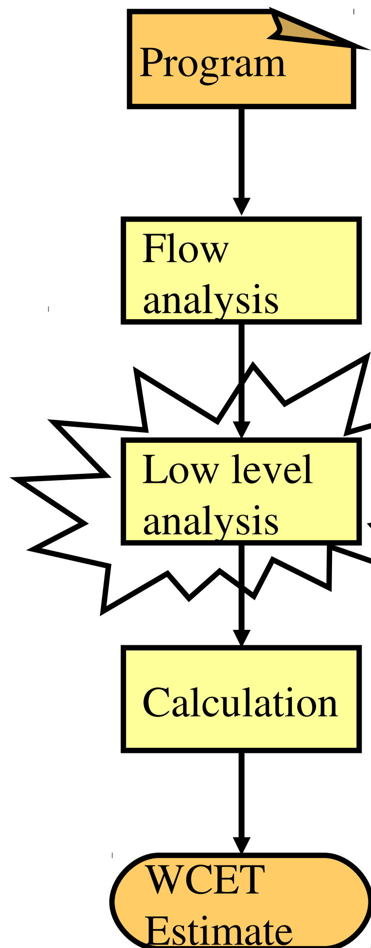
- **Infeasible path:**
  - Path A-B-C-E-F-G can not be executed
  - Since C implies ¬F
  - If (x > 5) then it is not possible that (x*2) < 0
- **Limits statically allowed executions**
  - Might tighten the WCET estimate

Example: Triangular Loop

```
   triangle(a,b):
A:    loop(i=1..100)
B:     loop(j=i..100)
C:        a[i,j]=...
        end loop
      end loop
```

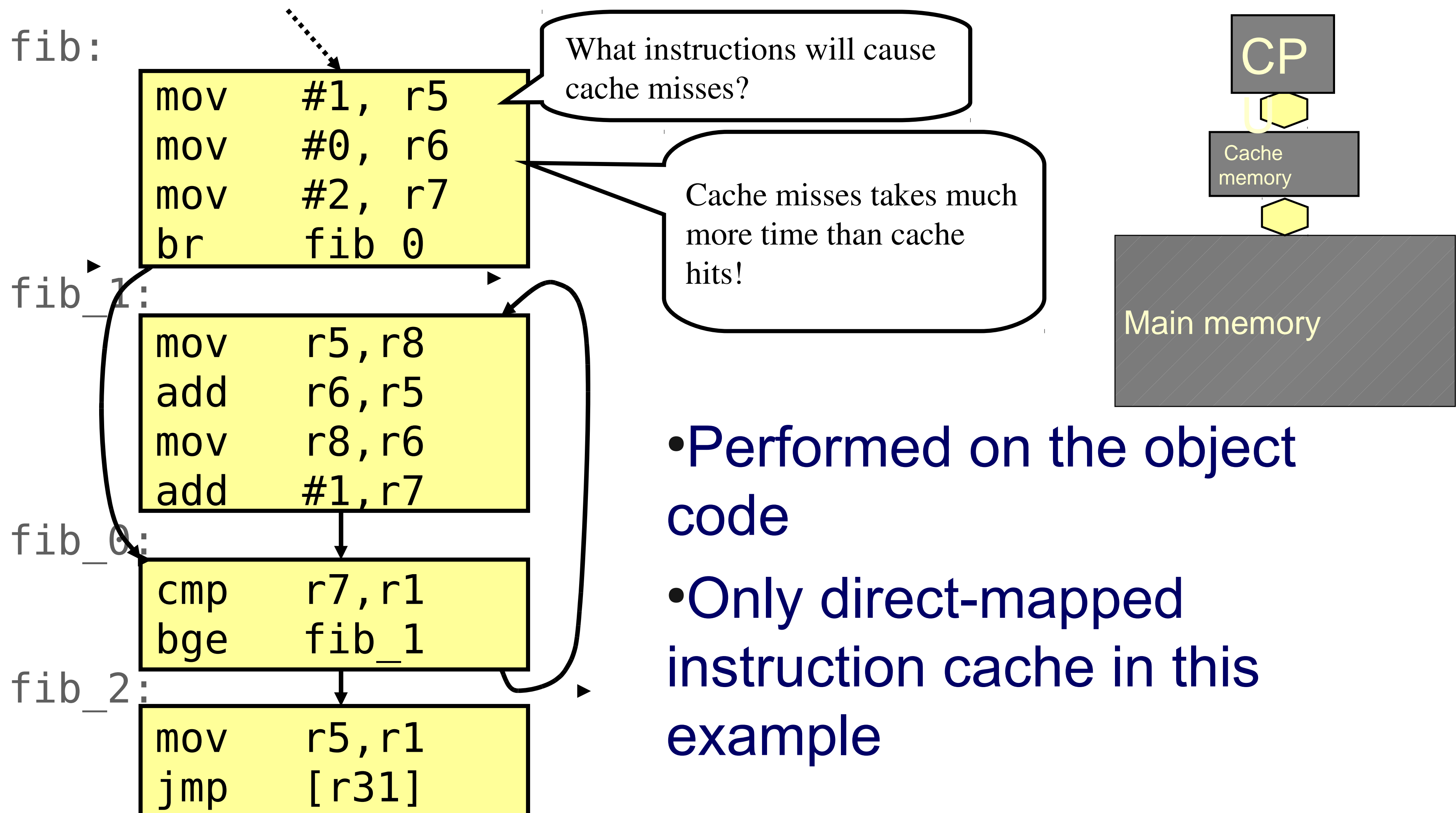
- Two loops:
  - Loop A bound: 100
  - Local B bound: 100
- Block C:
  - By loop bounds:
    100 * 100 = 10 000
  - But actually:
    100+...+1 = 5 050
- Limits statically allowed executions
  - Might tighten the WCET estimate

14



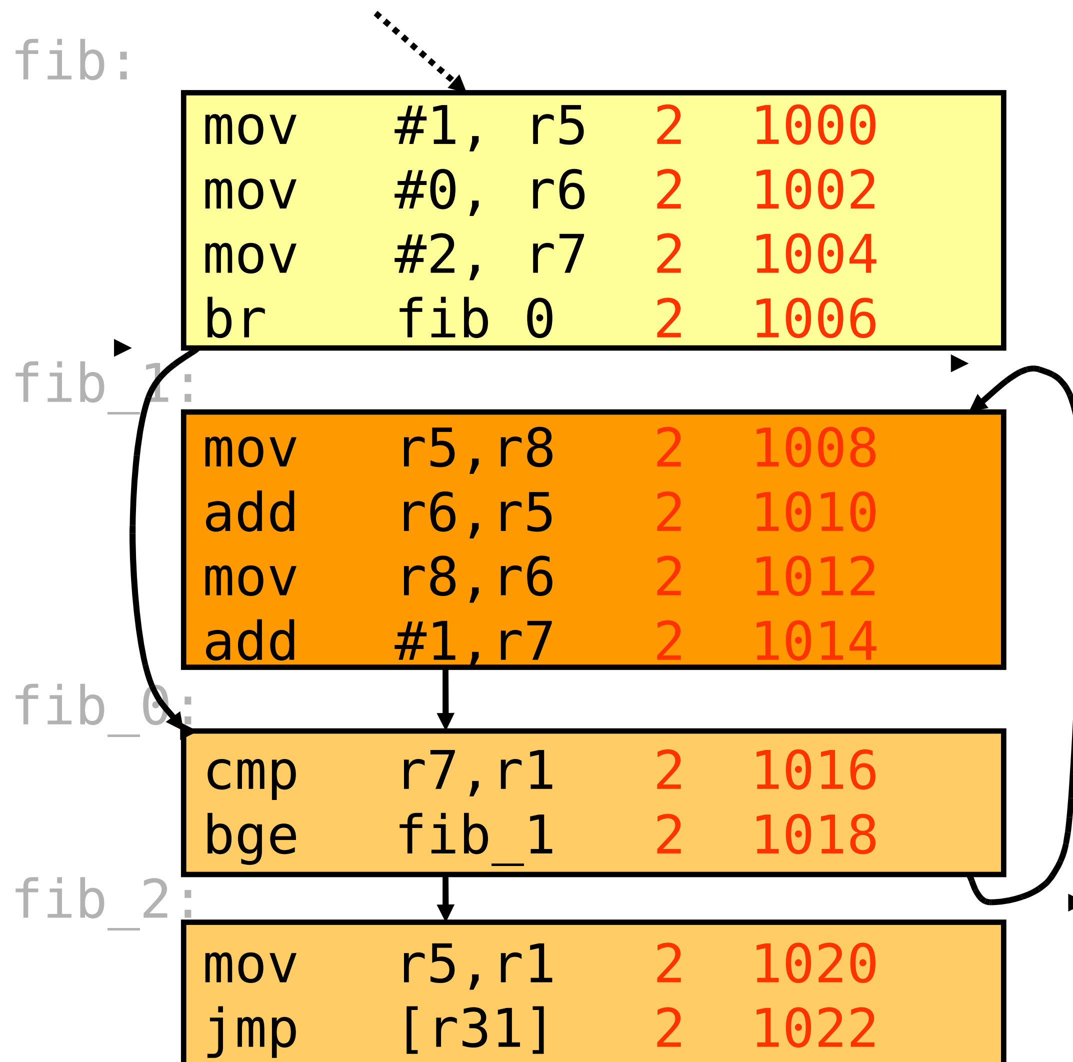
- Determine execution time bounds for program parts
  - Focus of most WCET-related research
- Using a <u>model</u> of the target HW
  - The model does not need to model all HW details
  - However, it should safely account for all possible HW timing effects
- Works on the binary, linked code
  - The executable program

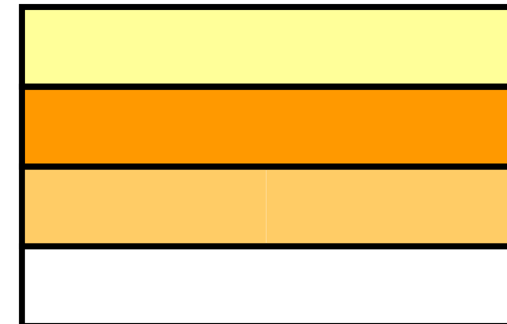
Example: Cache analysis



- Performed on the object code
- Only direct-mapped instruction cache in this example

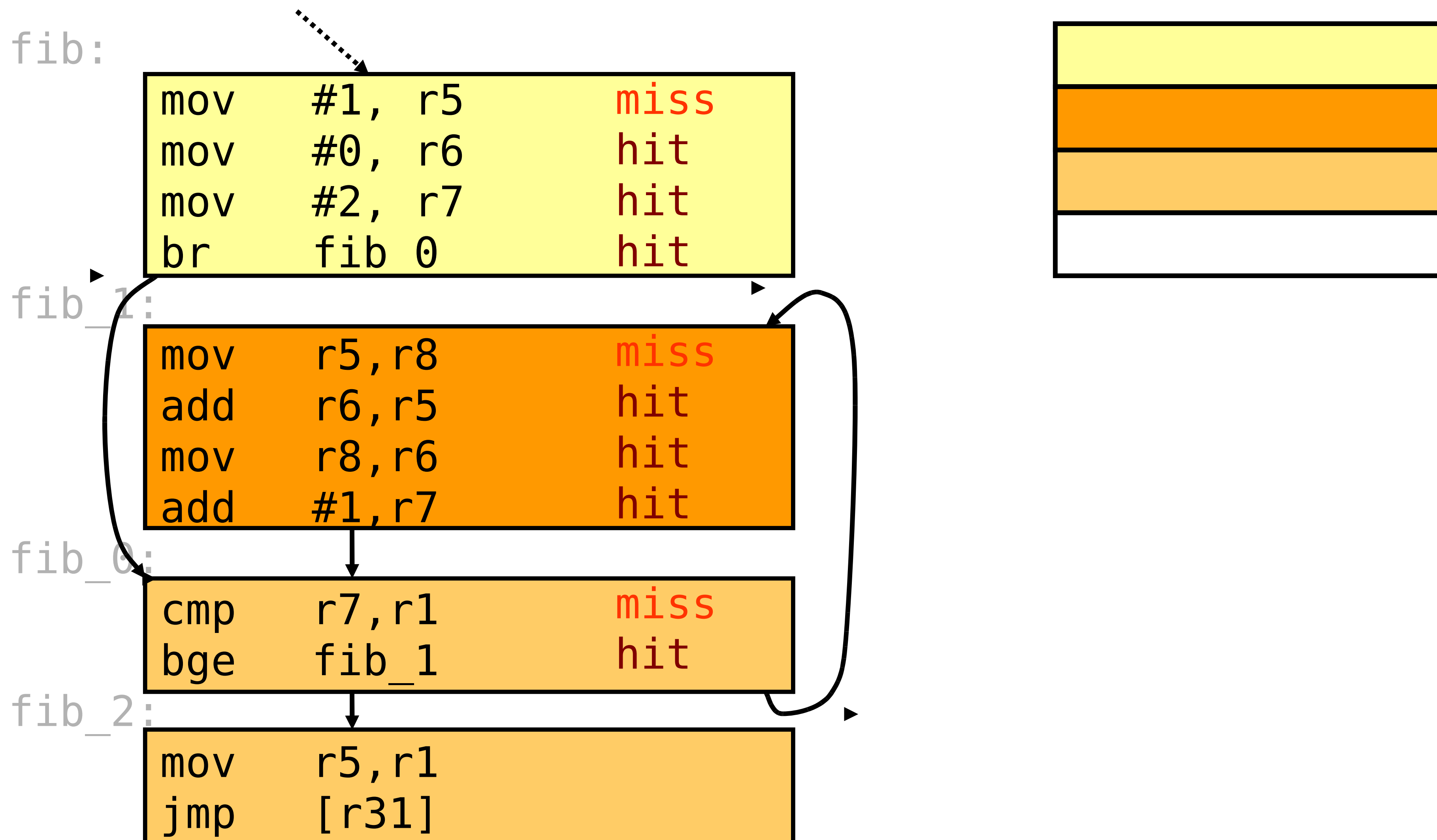
Example: Cache analysis



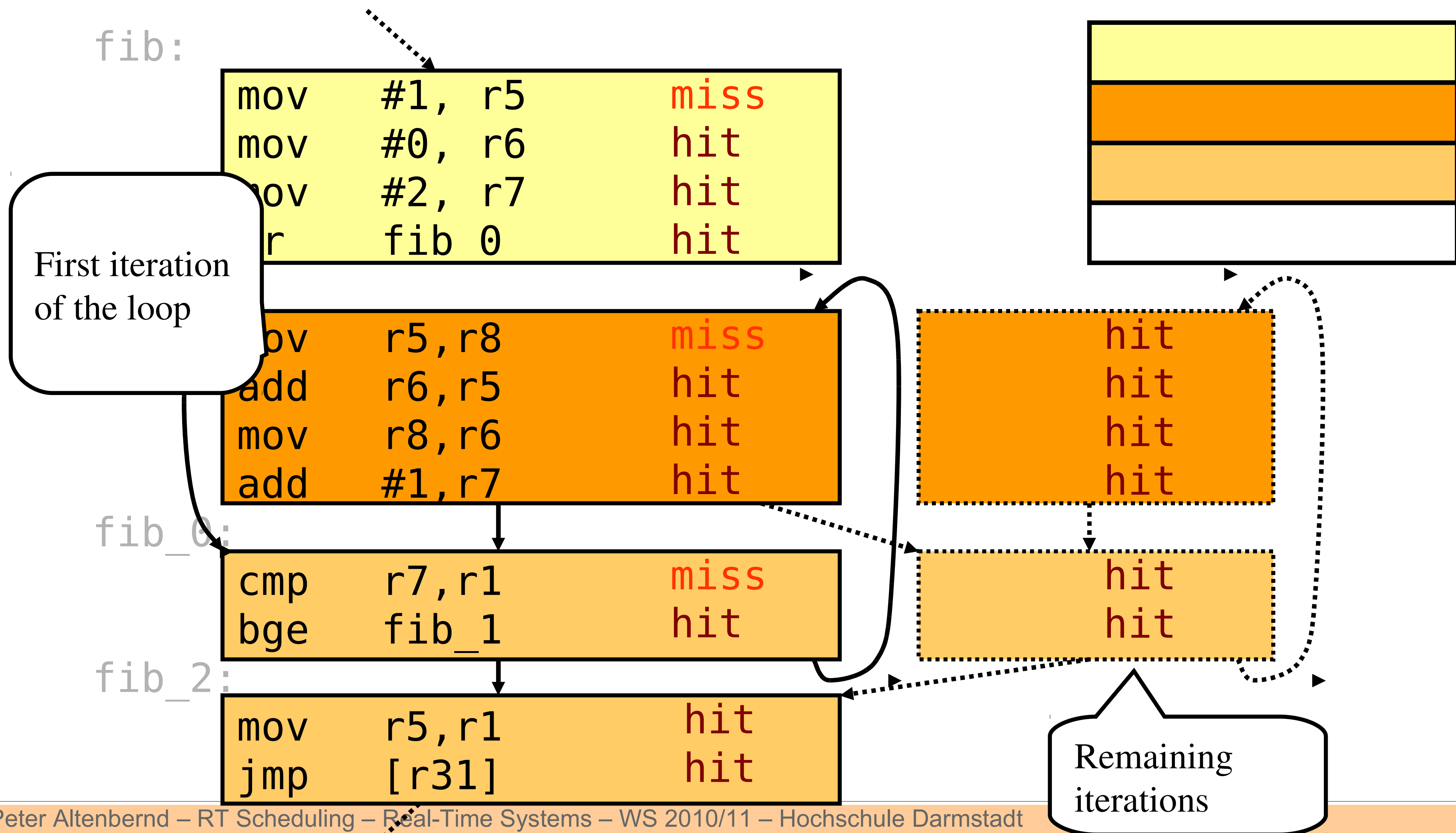


- Mapping to instruction cache

Example: Cache analysis

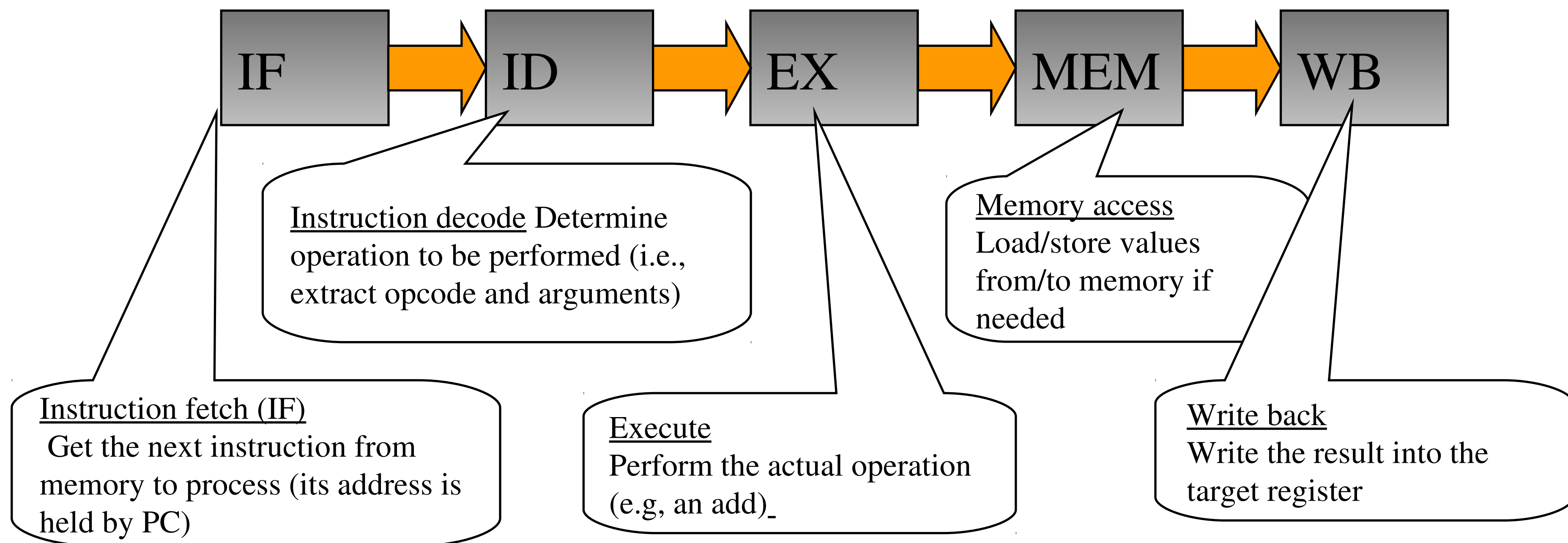


Example: Cache analysis



Example: CPU pipelines

- Observation: Most instructions go through same stages in the CPU
- Example: Classic RISC 5-stage pipeline



20

- Idea: Overlap the CPU stages of the instructions to achieve speed-up
- No pipelining:
  - Next instruction cannot start before previous one has finished all its stages

	1	2	3	4	5	6	7	8	9	10
IF										
ID										
EX										
MEM										
WB										

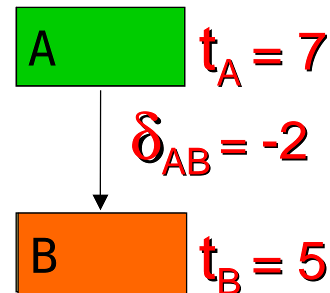
- Pipelining:
  - In principle: Speed-up = length of pipeline
  - However, often dependencies between instructions

	1	2	3	4	5	6
IF						
ID						
EX						
MEM						
WB						

21

Example: Simple Pipeline

```
    foo(x,i):
A:   while(i < 100)
B:     if (x > 5) then
C:       x = x*2;
       else
D:       x = x+2;
       end
E:     if (x < 0) then
F:       b[i] = a[i];
       end
G:     i = i+1;
   end
```



$t_A = 7$

$\delta_{AB} = -2$

$t_B = 5$

	1	2	3	4	5	6	7
IF							
EX							
M							
F							

	1	2	3	4	5
IF					
EX					
M					
F					

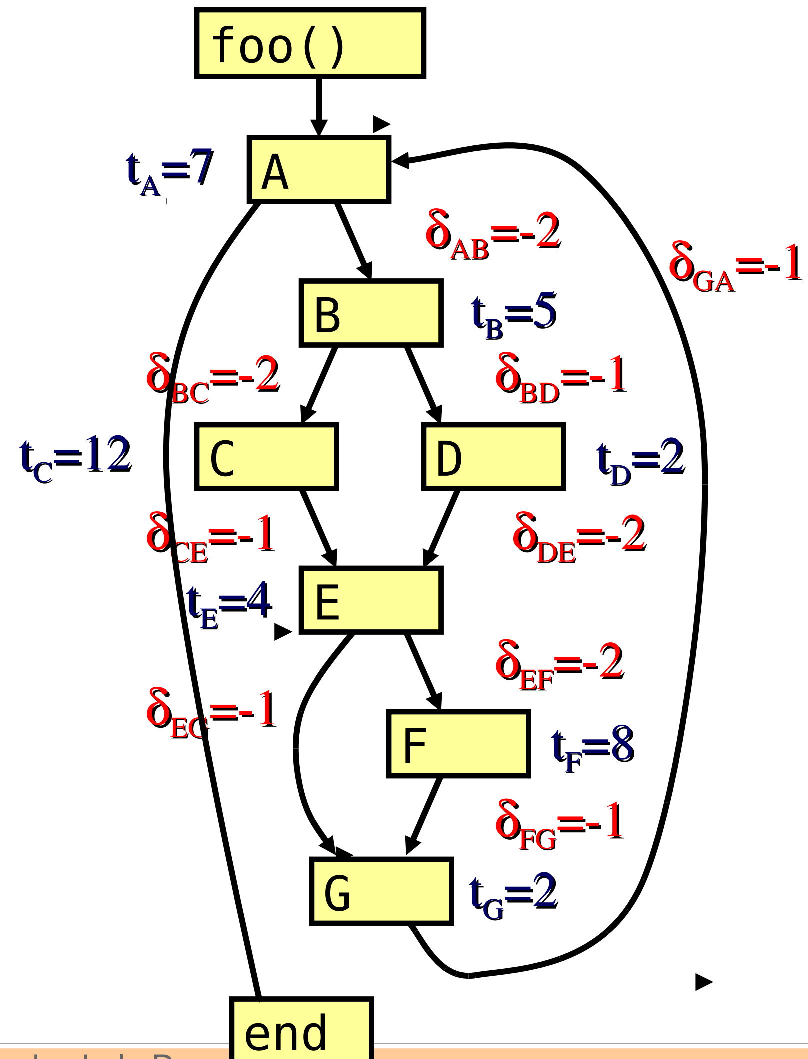
	1	2	3	4	5	6	7	8	9	10
IF										
EX										
M										
F										

$t_{AB} = 10$

$\delta_{AB} = 10 - (7 + 5) = -2$

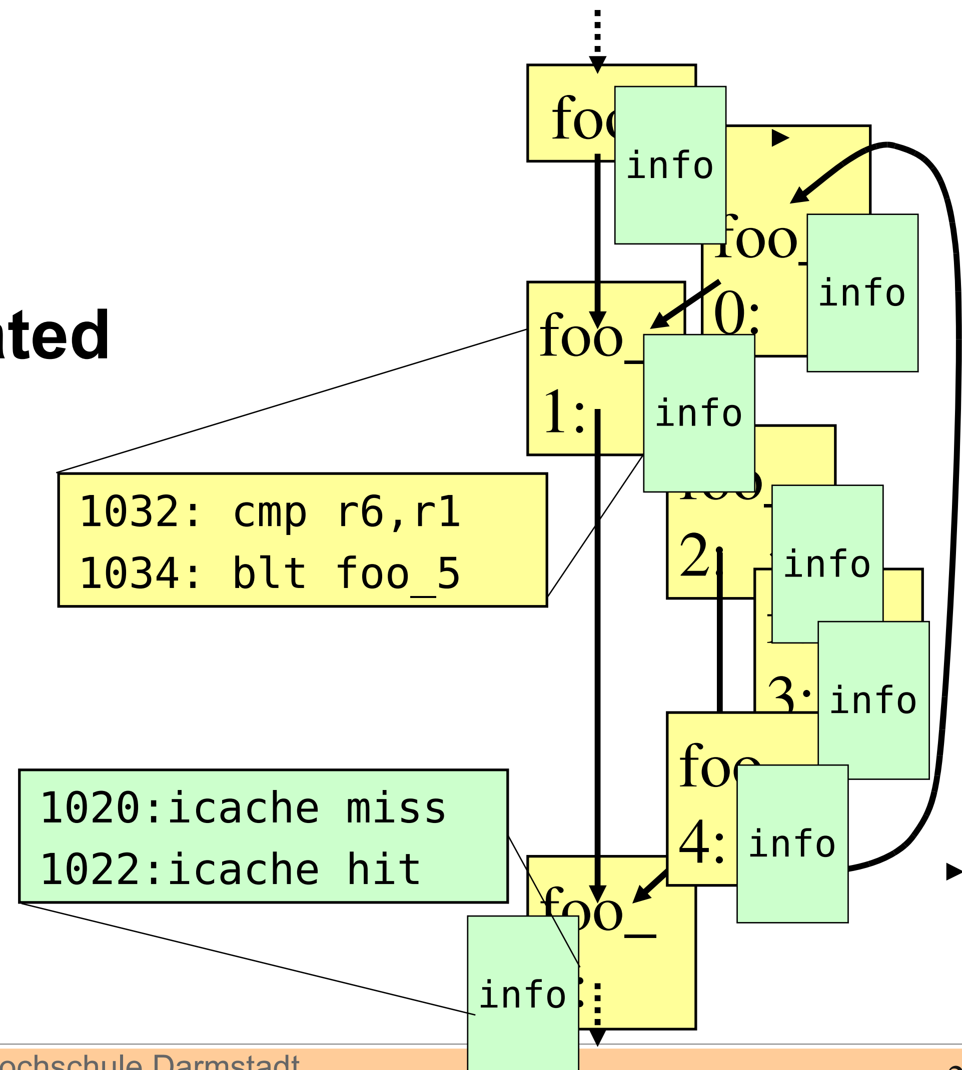
Example: Pipeline result

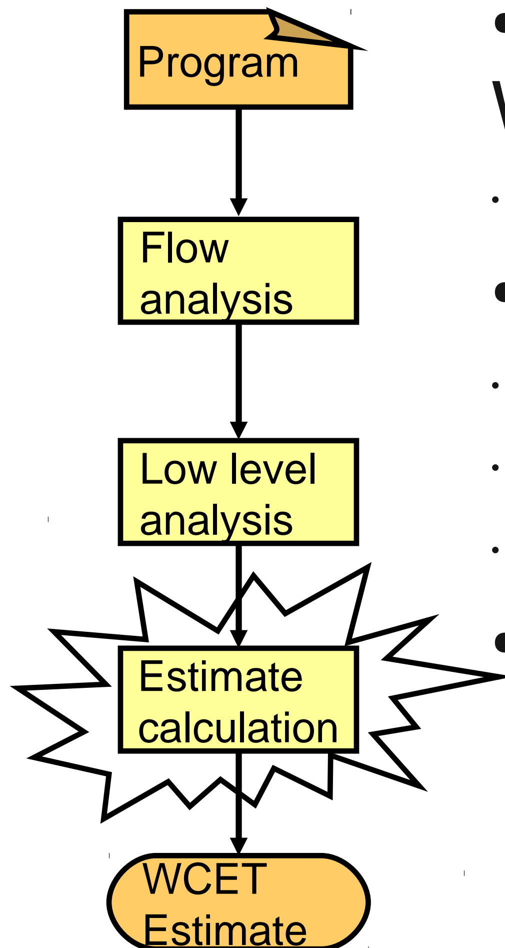
```
    foo(x,i):
A:   while(i < 100)
B:     if (x > 5) then
C:       x = x*2;
       else
D:       x = x+2;
       end
E:     if (x < 0) then
F:       b[i] = a[i];
       end
G:     i = i+1;
     end
```



Cache & Pipeline analysis

- Pipeline analysis might take cache analysis results as input
  - **Instructions gets annotated with cache hit/miss**
  - **These misses/hits affect pipeline timing**
- Complex HW require integrated cache & pipeline analysis

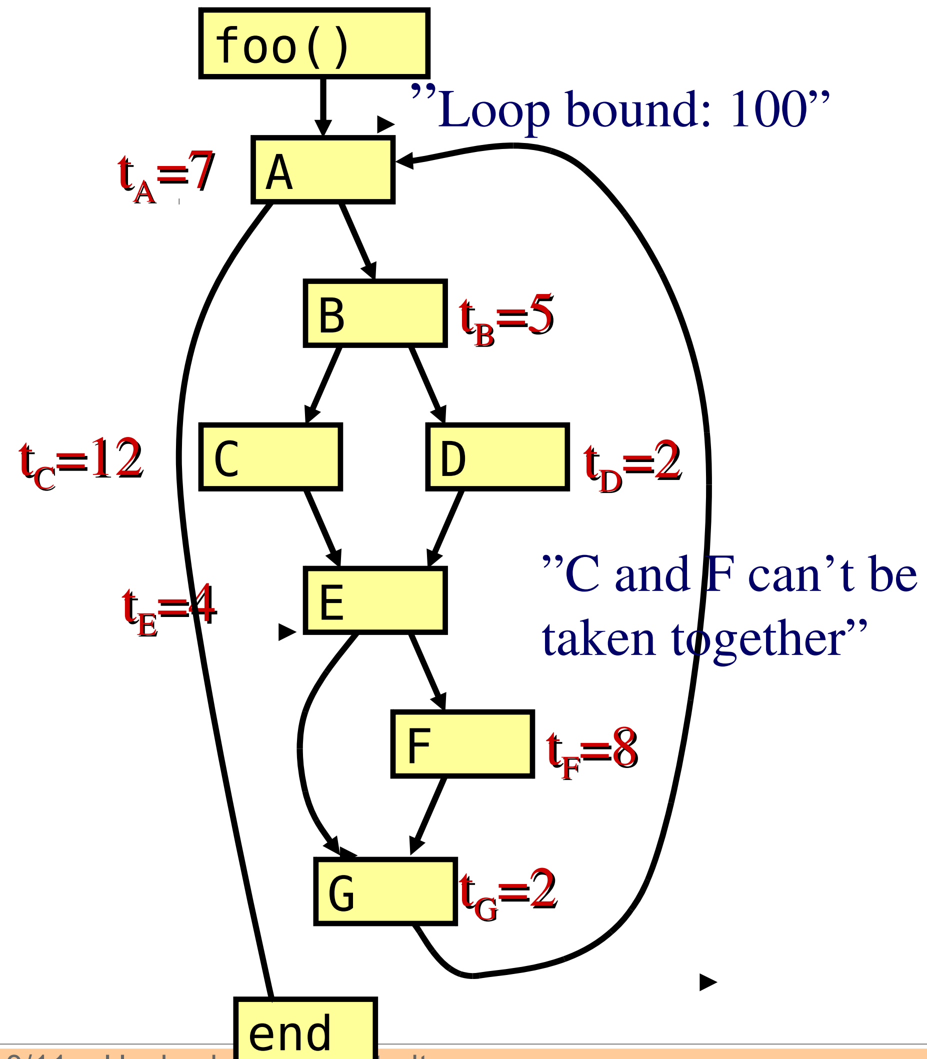




- Derive an upper bound on the program's WCET
  - Given flow and timing information
- Several approaches used:
  - Tree-based
  - Path-based
  - Constraint-based (IPET)
- Properties of approaches:
  - ☐ Flow information handled
  - ☐ Object code structure allowed
  - ☐ Modeling of hardware timing
  - ☐ Solution complexity

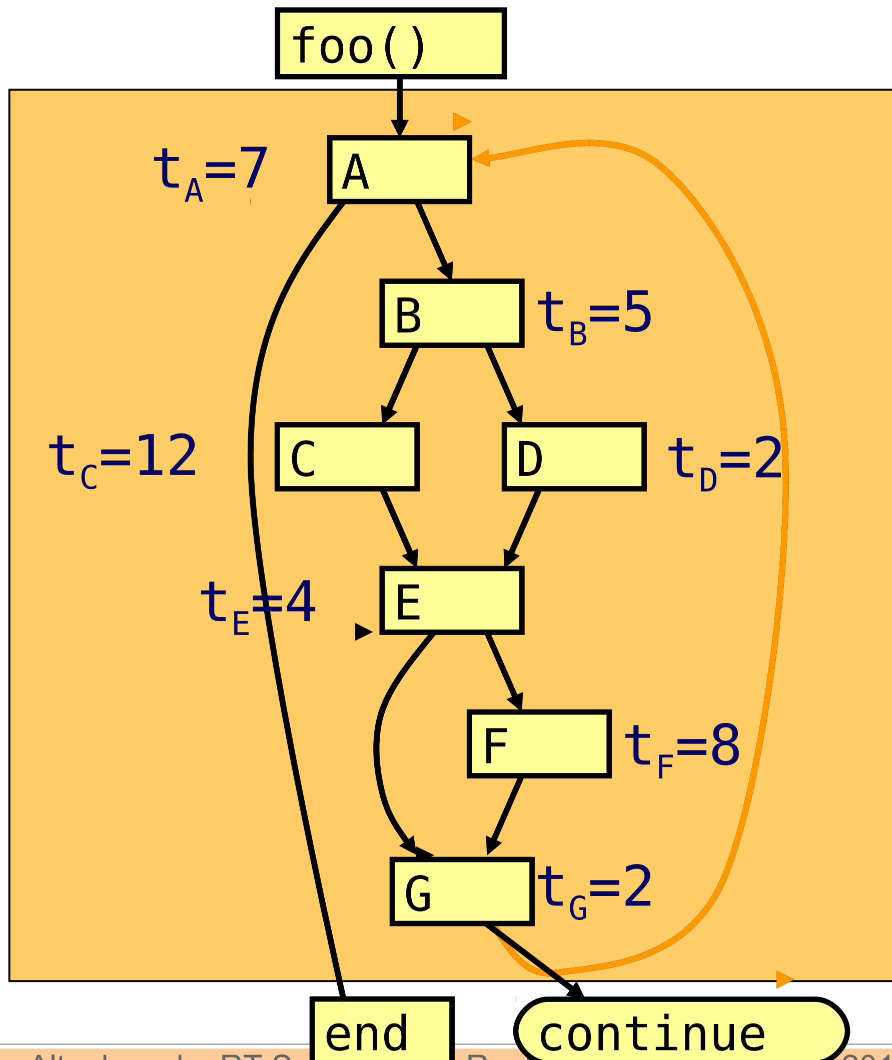
Example: Combined flow analysis and low-level analysis result

```
    foo(x,i):
A:   while(i < 100)
B:     if (x > 5) then
C:       x = x*2;
       else
D:       x = x+2;
       end
E:     if (x < 0) then
F:       b[i] = a[i];
       end
G:     i = i+1;
     end
```



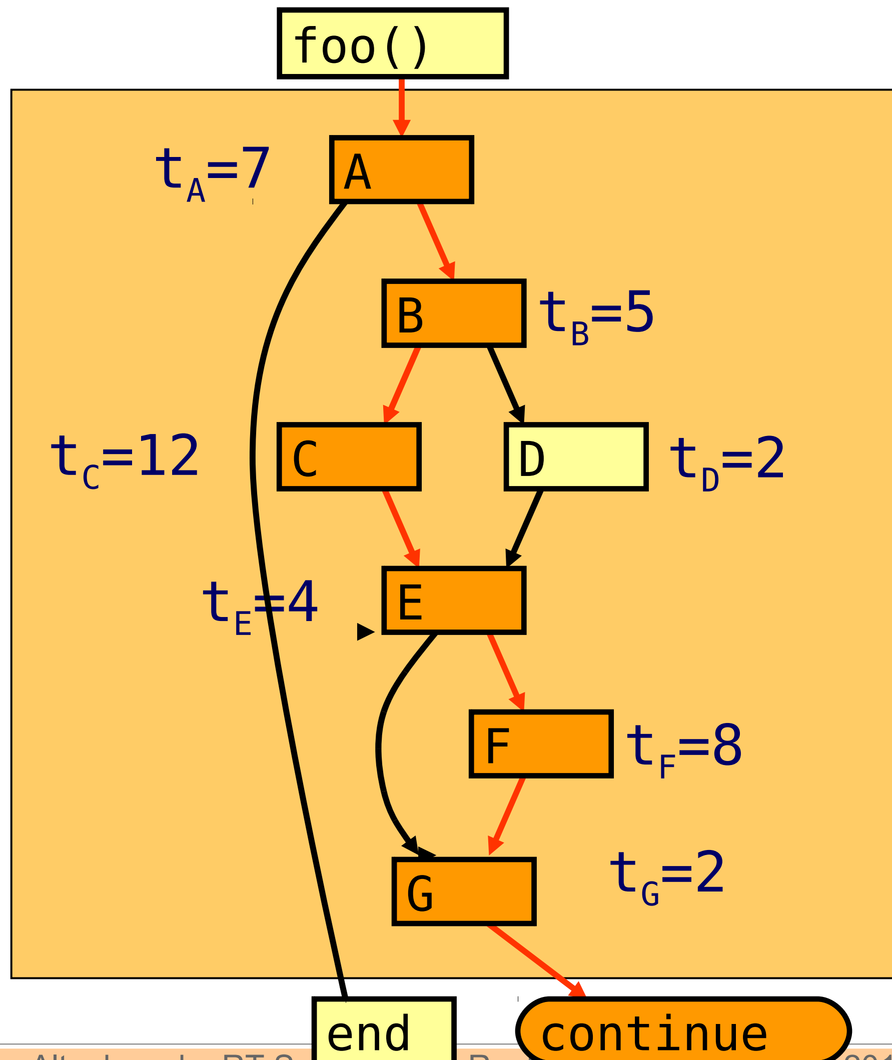
Path-Based Ca

```
    foo(x,i):
A:   while(i < 100)
B:     if (x > 5) then
C:       x = x*2;
       else
D:       x = x+2;
       end
E:     if (x < 0) then
F:       b[i] = a[i];
       end
G:     i = i+1;
     end
```



- **Find longest path**
  - One loop at a time
- **Prepare the loop**
  - Remove back edges
  - Redirect to special continue nodes

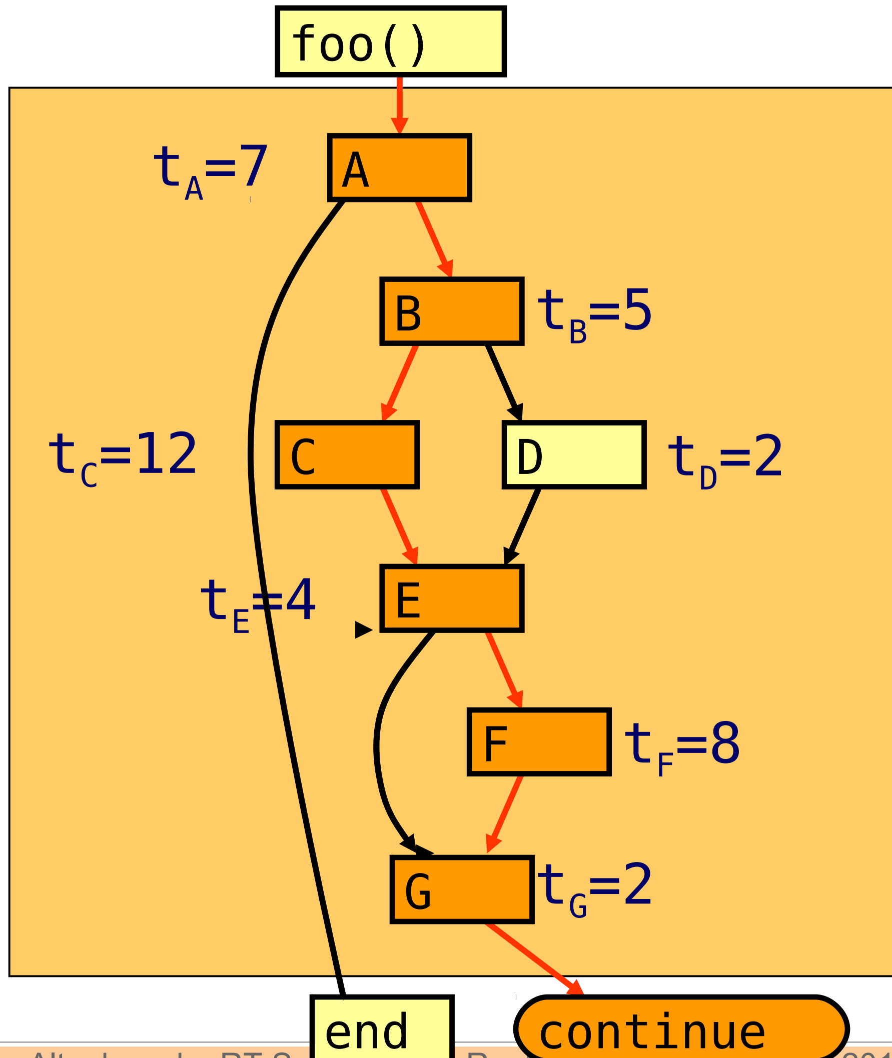
Path-Based Calculation



- Longest path:
  - A-B-C-E-F-G
  - 7+5+12+4+8+2= 38 cycles
- Total time:
  - 100 iterations
  - 38 cycles per iteration
  - Total: 3800 cycles

Path-Based Calc...

```
       foo(x,i):
A:     while(i < 100)
B:       if (x > 5) then
C:         x = x*2;
         else
D:         x = x+2;
         end
E:       if (x < 0) then
F:         b[i] = a[i];
         end
G:       i = i+1;
       end
```

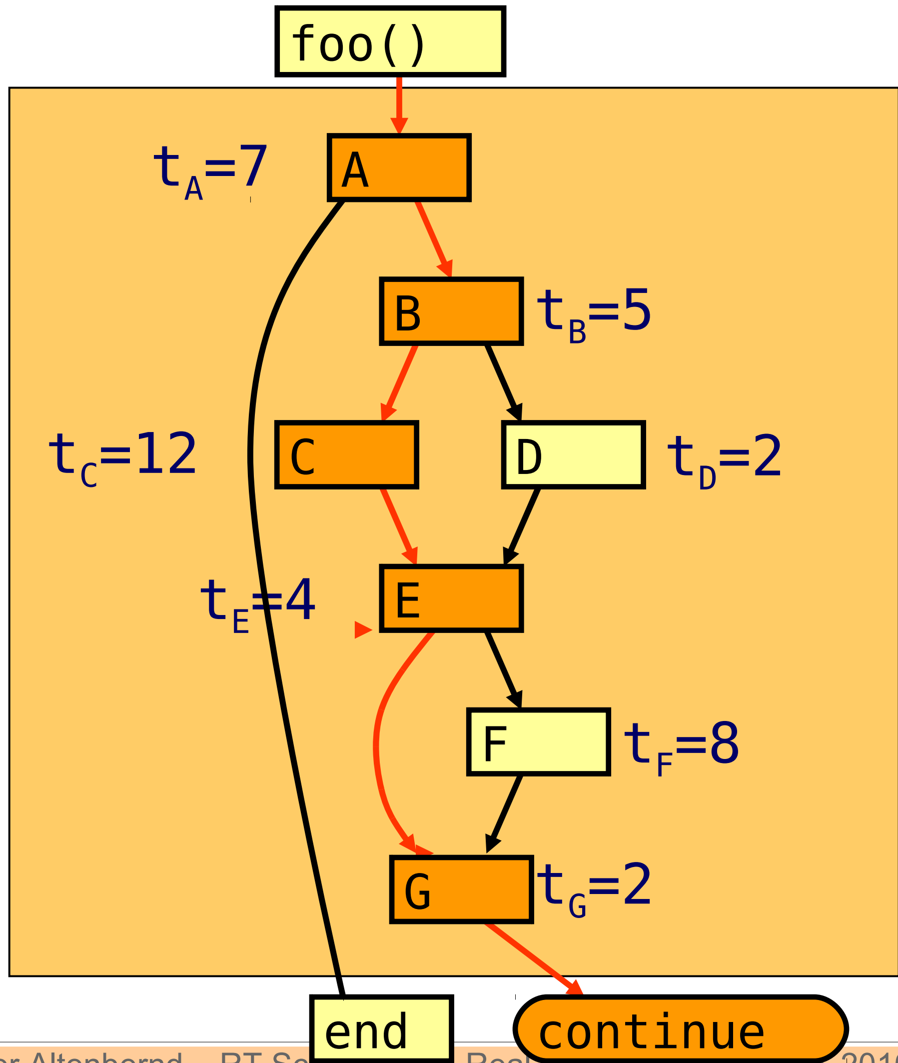


C and F can never execute together

- **Infeasible path:**
  - A-B-C-E-F-G
  - Ignore, look for next

Path-Based Calc

```
    foo(x,i):
A:   while(i < 100)
B:     if (x > 5) then
C:       x = x*2;
       else
D:       x = x+2;
       end
E:     if (x < 0) then
F:       b[i] = a[i];
       end
G:     i = i+1;
     end
```



C and F can never execute together

- Infeasible path:
  - A-B-C-E-F-G
  - Ignore, look for next
- New longest path:
  - A-B-C-E-G
  - 30 cycles
- Total time:
  - Total: 3000 cycles