Worst-Case Execution Time Analysis

Andreas Ermedahl, Associate Professor
Mälardalen Real-Time Research Center (MRTC)
Västerås, Sweden
andreas.ermedahl@mdh.se

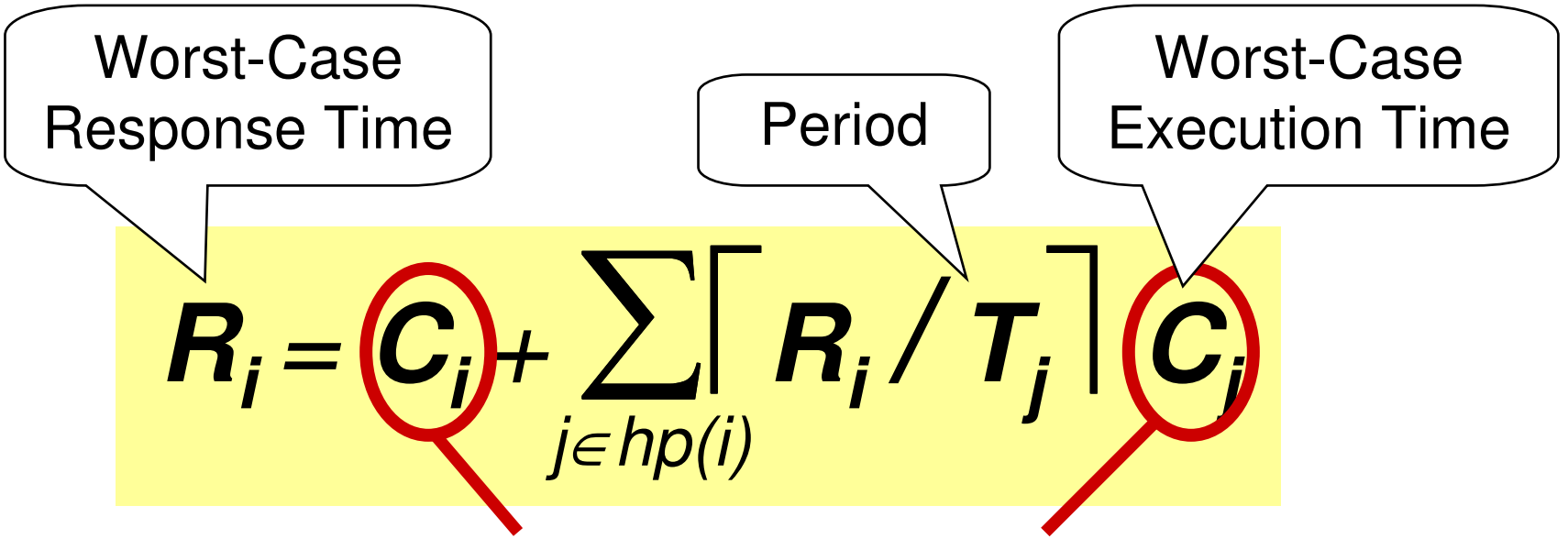
MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY


or...
The search for
the missing
C

What C are we talking about?

- A key component in the analysis of real-time systems
- You have seen it in formulas such as:



Where do these C values come from?

Program timing is not trivial!

```
int f(int x) {
    return 2 * x;
}
```

Simpler questions

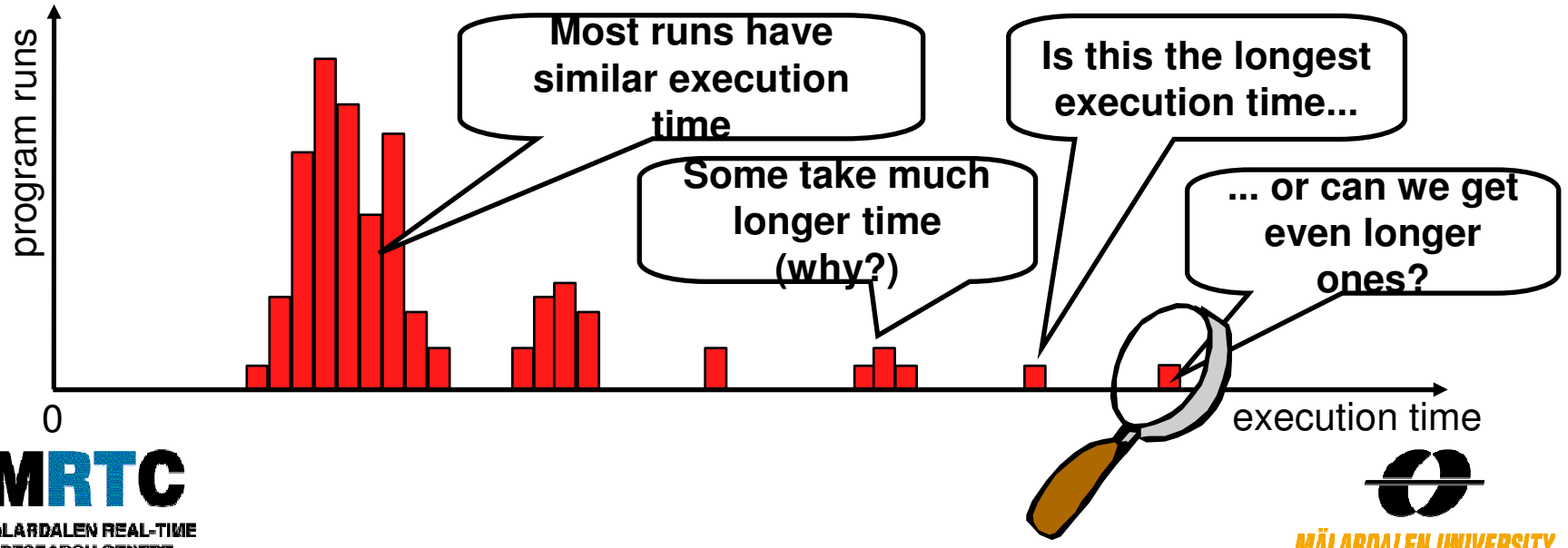
* What is the program doing?
* Will it always do the same thing?
* How important is the result?

Harder questions

* What is the execution time of the program?
* Will it always take the same time to execute?
* How important is execution time?

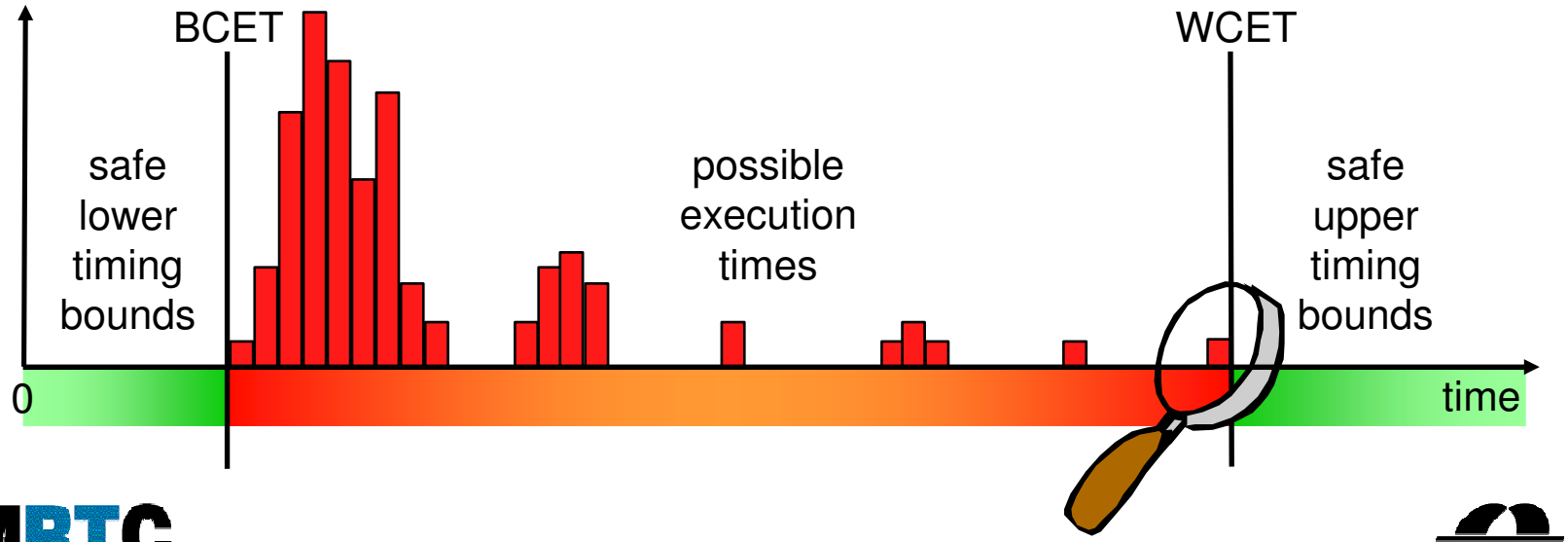
Program timing basics

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



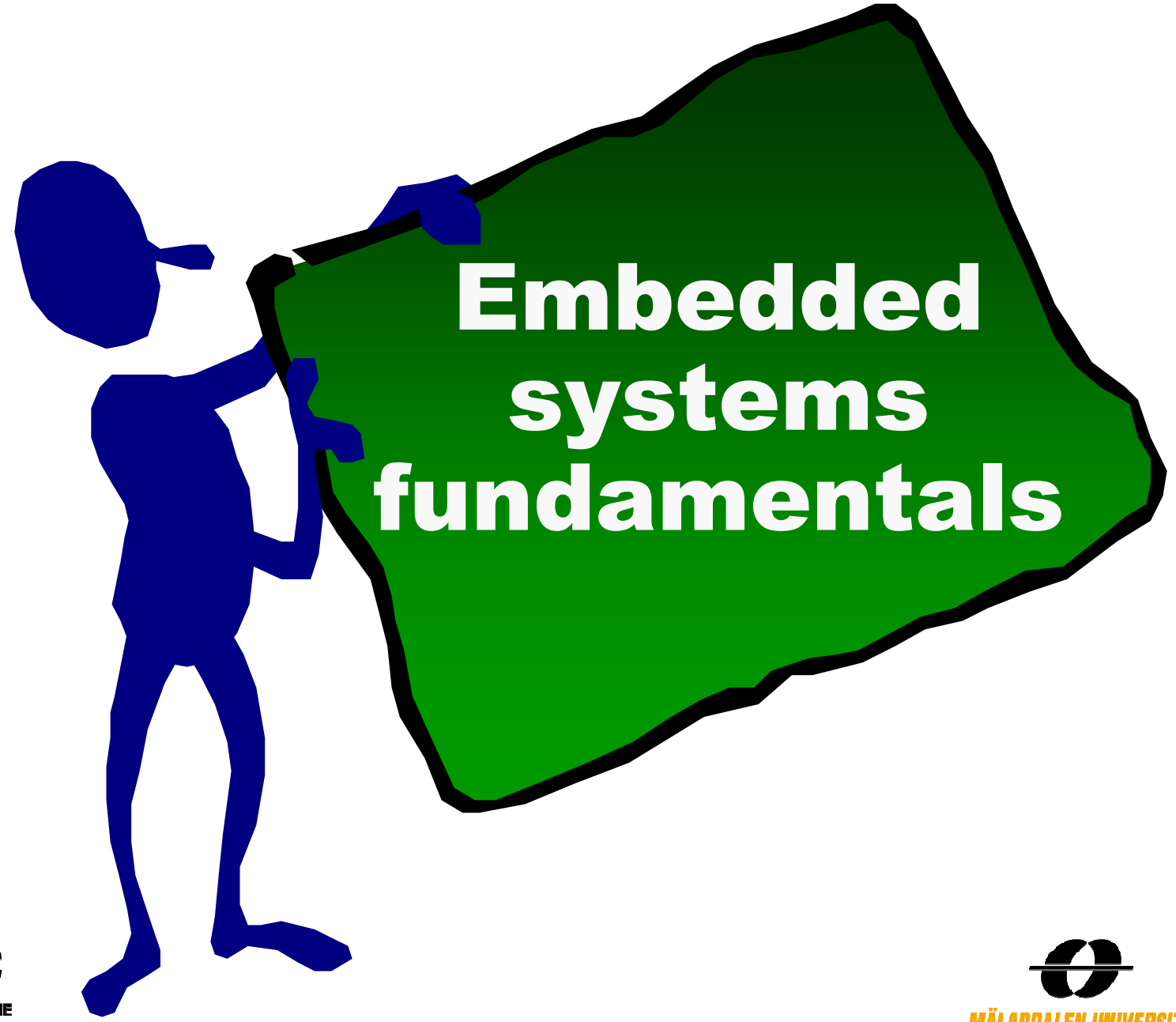
WCET and WCET analysis

* **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 safe upper bound on a program's WCET**



Presentation outline

- Embedded system fundamentals
- WCET analysis
  - Measurements
  - Static analysis
  - Flow analysis, low-level analysis, and calculation
  - Hybrid approaches
- WCET analysis tools
- The SWEET approach to WCET analysis
- Multi-core + WCET analysis?
- WCET analysis assignment



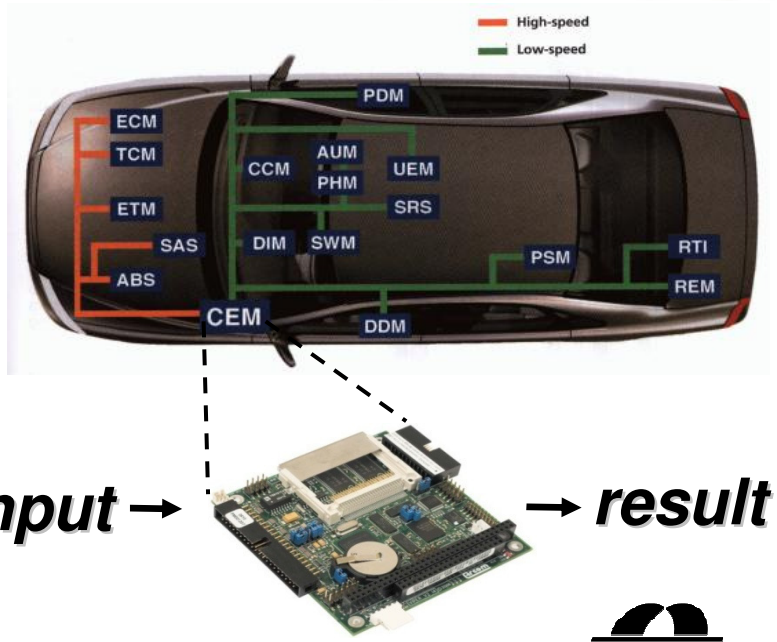
MRTC
MÄLARDALEN REAL-TIME
RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Embedded computers

- An integrated part of a larger system
  - Example: A microwave oven contain at least one embedded processor
  - Example: A modern car can contain more than 100 embedded processors

- Interacts with the user, the environment, and with other computers
  - Often limited or no user interface
  - Often with timing constraints



MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Embedded systems everywhere

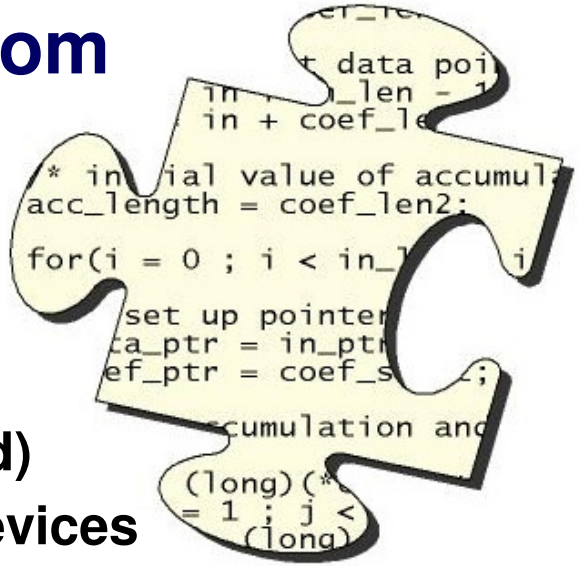
- **Today, all advanced products contain embedded computers!**
  - **Our society is dependant on that they function correctly**

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

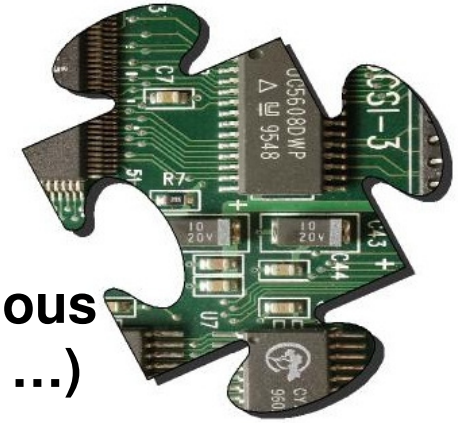
Embedded systems software

- Amount of software can vary from extremely small to very large
  - Gives characteristics to the product
- Often developed with target hardware in mind
  - Often limited resources (memory / speed)
  - Often direct accesses to different HW devices
  - Not always easily portable to other HW
- Many different programming languages
  - C still dominates, but often special purpose languages
- Many different software development tools
  - Not just GCC and/or Microsoft Visual Studio

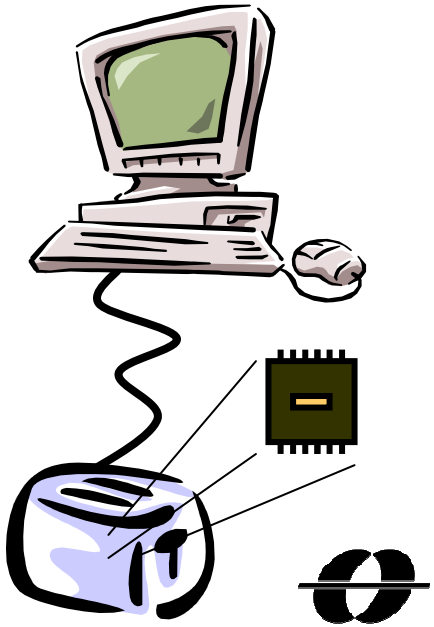


Embedded system hardware

* Huge variety of embedded system processors
  - Not just one main processor type as for PCs
  - Additionally, same CPU can be used with various hardware configurations (memories, devices, …)



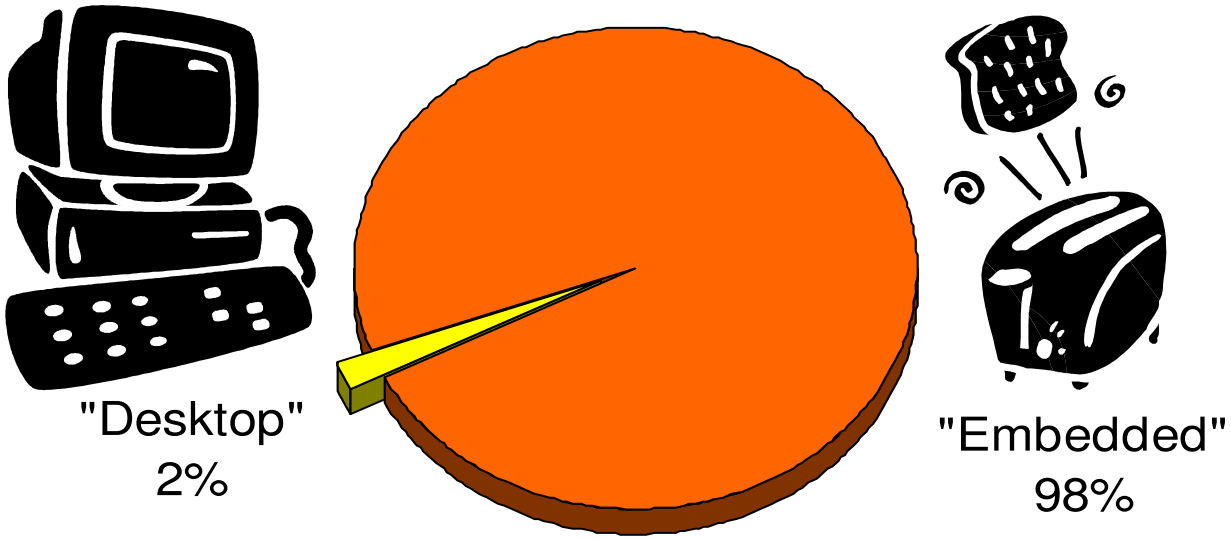
* The hardware is often tailored specifically to the application
  - E.g., using a DSP processor for signal processing in a mobile telephone



* Cross-platform development
  - E.g., develop on PC and download final application to target HW

Some interesting figures

- 4 billion embedded processors sold in 2008
  - Global market worth €60 billion
  - Predicted annual growth rate of 14%
  - Forecasts predict more than 40 billion embedded devices in 2020
- Embedded processors clearly dominate yearly production



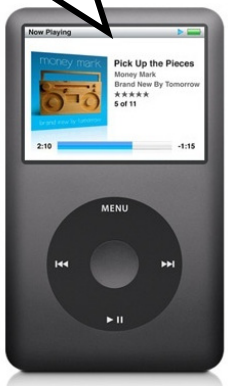
Source: http://www.artemis-ju.eu/embedded_systems

Real-time systems

- **Computer systems where the timely behavior is a central part of the function**
  - **Containing one or more embedded computers**
  - **Both soft- and hard real-time, or a mixture…**

Timing of radio communication, speech recognition,…

Timing of music playing from MP3 file



Timing of radio communication, motor control, rudder and flaps control,…

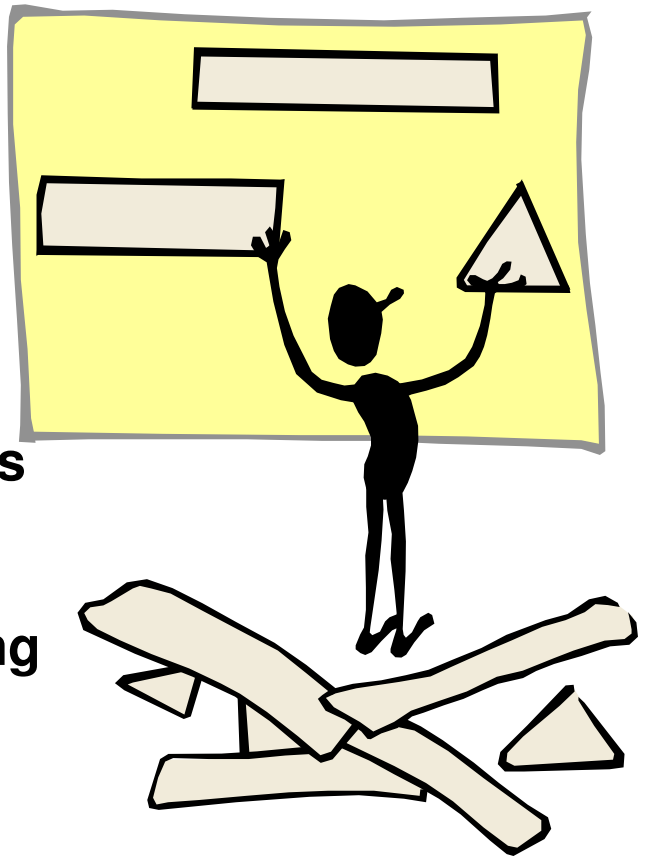
Timing of network communication, motor control, ABS brakes, anti-slip control,…

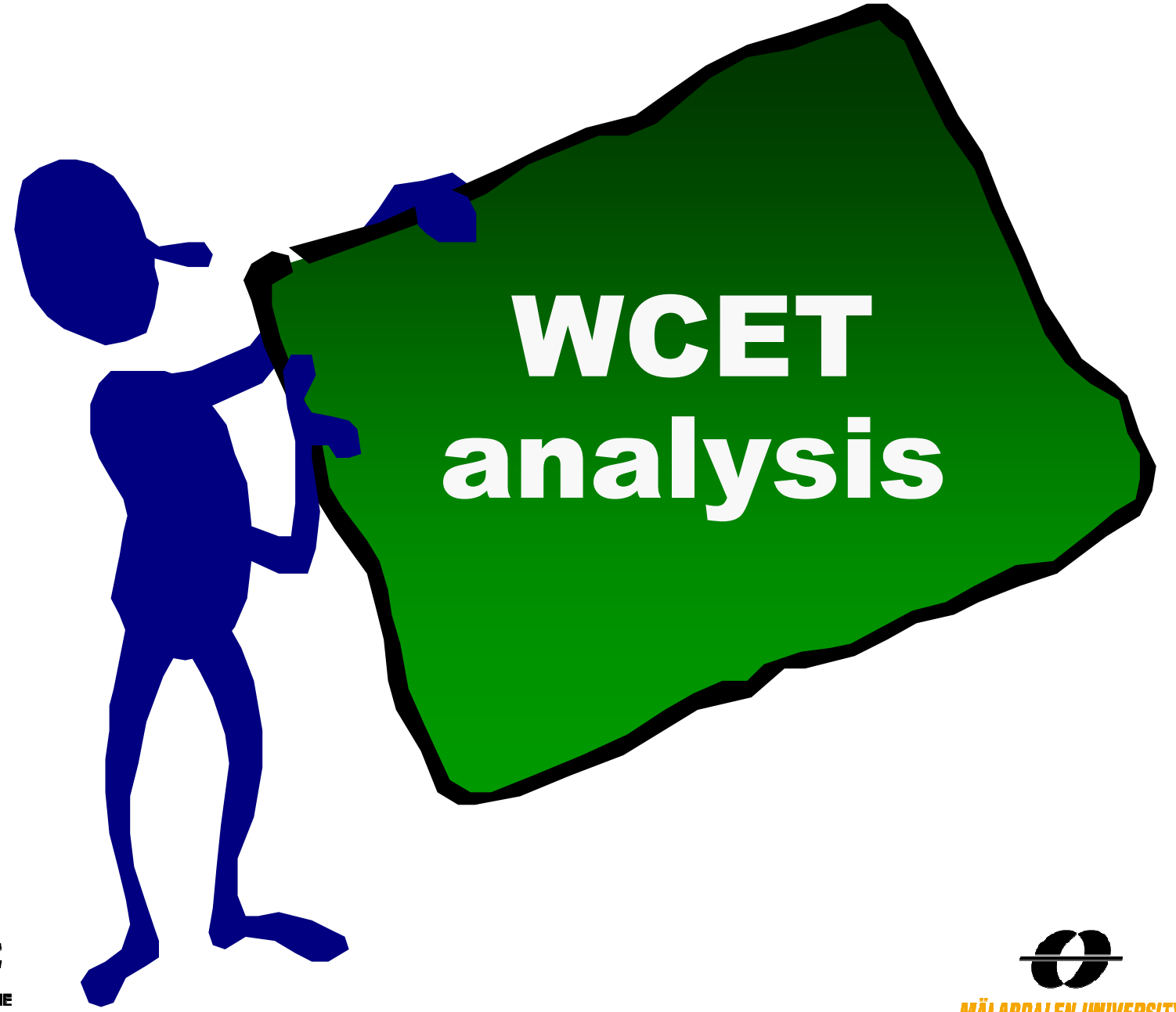
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Uses of reliable WCET bounds

- **Hard real-time systems**
  - WCET needed to guarantee behavior
- **Real-time scheduling**
  - Creating and verifying schedules
  - Large part of RT research assume the existence of reliable WCET bounds
- **Soft real-time systems**
  - WCET useful for system understanding
- **Program tuning**
  - Critical loops and paths
- **Interrupt latency checking**



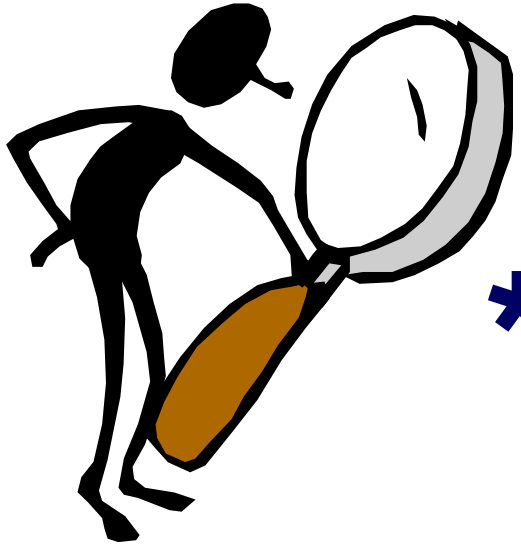


MRTC
MÄLARDALEN REAL-TIME
RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Obtaining WCET bounds

- **Measurement**
  - Industrial practice



- **Static analysis**
  - Research front

Measuring for the WCET

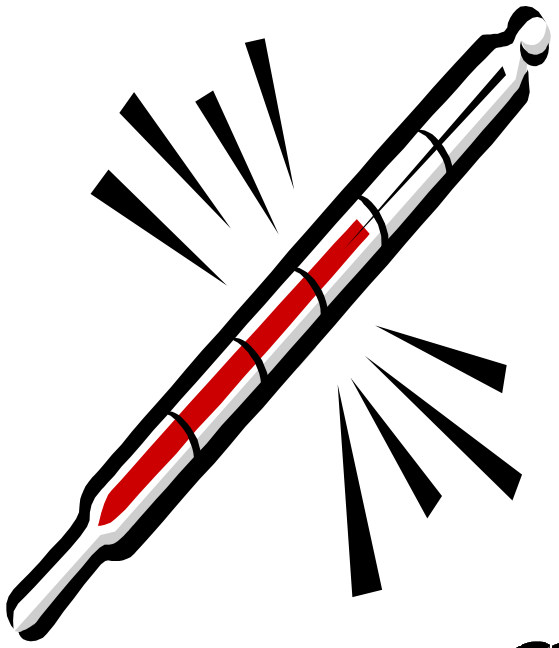
- **Methodology:**
  - Determine potential ”worst-case input”
  - Run and measure
  - Add a safety margin

Measurement issues

- 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?

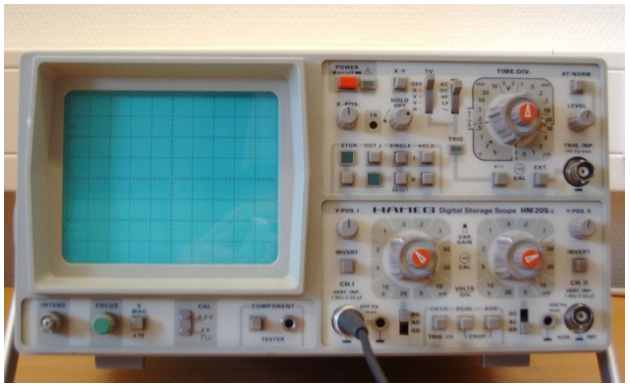
SW measurement methods

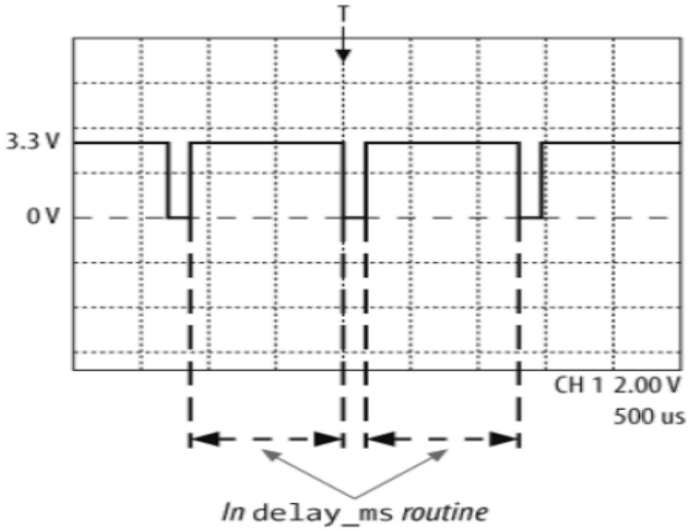
- **Operating system facilities**
  - Commands such as `time`, `date` and `clock`
  - Note that all OS-based solutions require precise HW timing facilities (and an OS)
- **Cycle-level simulators**
  - Software simulating CPU
  - Correctness vs. hardware?
- **High-water marking**
  - Keep system running
  - Record maximum time observed for task
  - Keep in shipping systems, read at service intervals



Using an oscilloscope

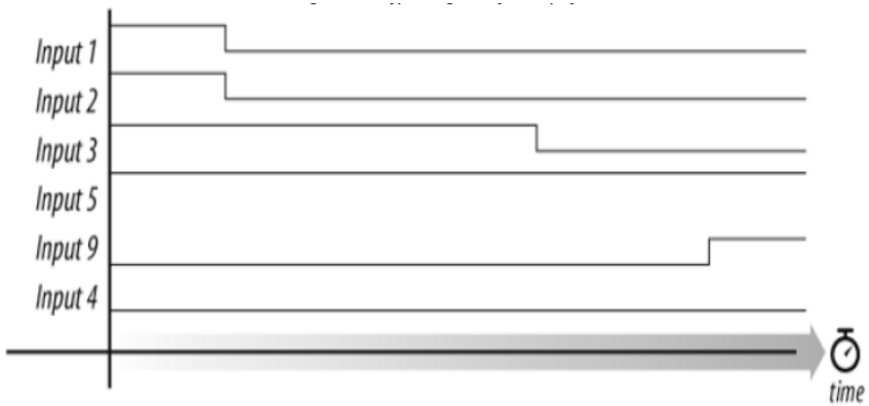
- Common equipment for HW debugging
  - Used to examine electrical output signals of HW
- Observes the voltage or signal waveform on a particular pin
  - Usually only two to four inputs
- To measure time spent in a routine:
  1. Set I/O pin high when entering routine
  2. Set the same I/O pin low before exiting
  3. Oscilloscope measures the amount of time that the I/O pin is high
  4. This is the time spent in the routine

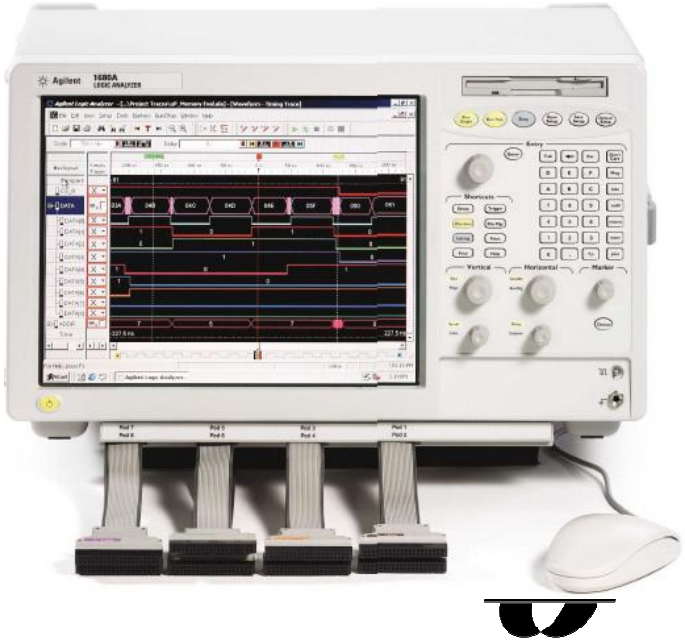




Using a logic analyzer

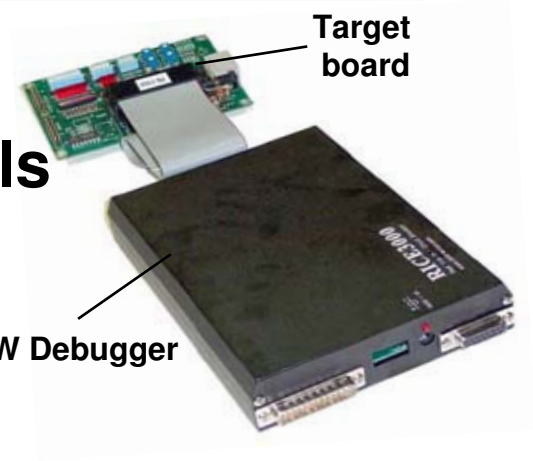
* Equipment designed for troubleshooting digital hardware
* Have dozens or even hundreds of inputs
  - Each one keeping track on whether the electrical signal it is attached to is currently at logic level 1 or 0
  - Result can be displayed against a timeline
  - Can be programmed to start capturing data at particular input patterns

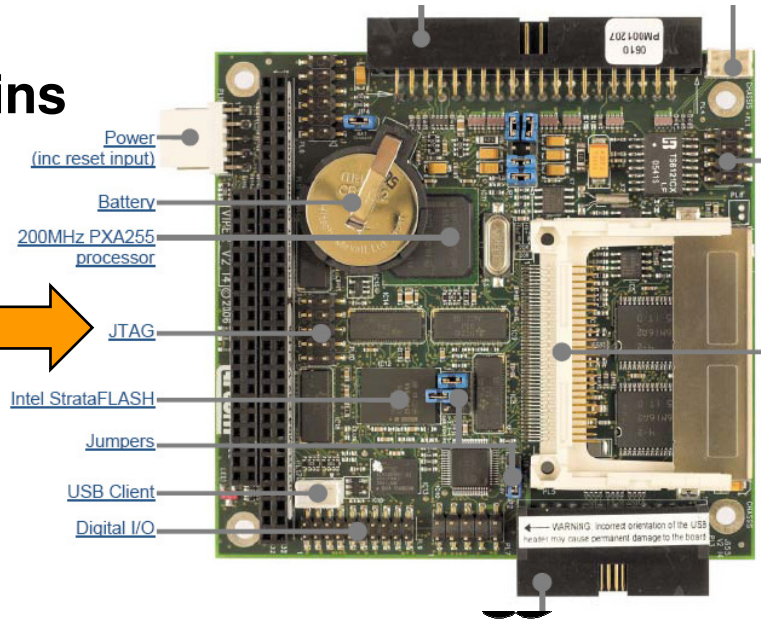




HW measurement tools

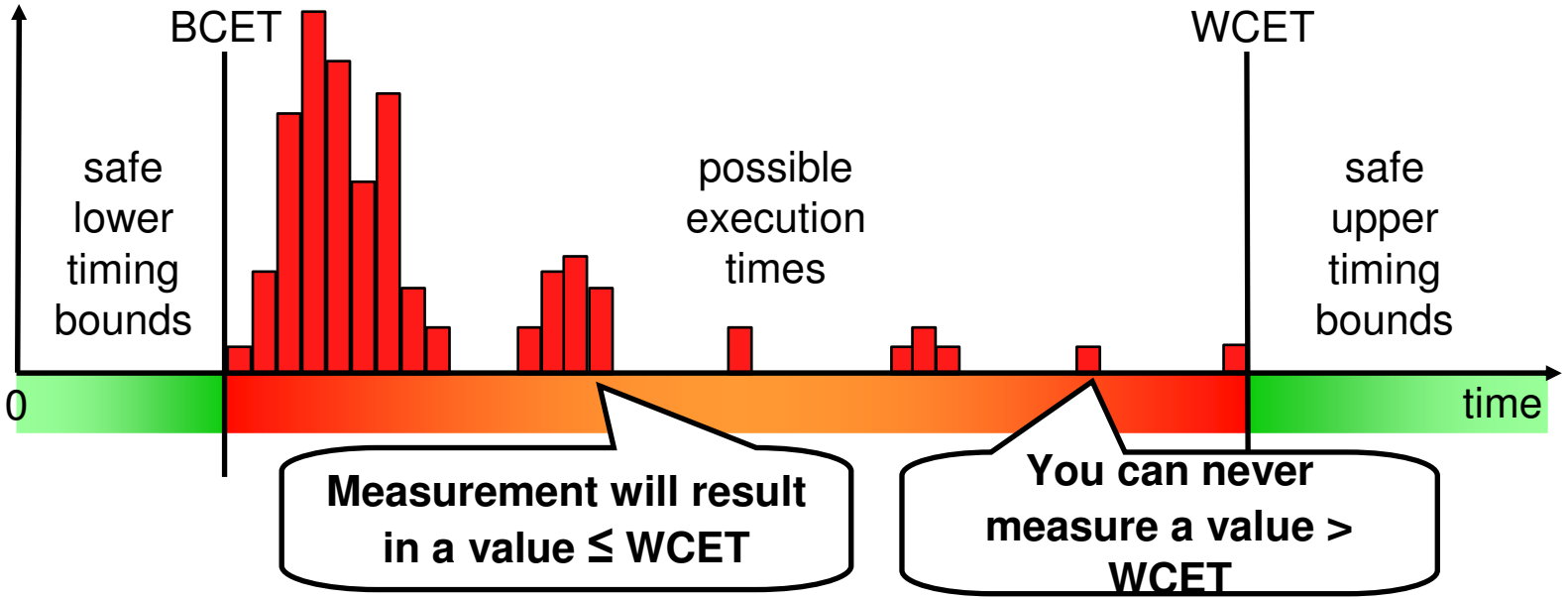
- **In-circuit emulators (ICE)**
  - Special CPU version revealing internals
    - ➨High visibility & bandwidth
    - ➨High cost + supportive HW required
- **Processors with debug support**
  - Designed into processor
    - ➨Use a few dedicated processor pins
  - Using standardized interfaces
    - ➨Nexus debug interfaces, JTAG, Embedded Trace Macrocell, …
  - Supportive SW & HW required
  - Common on modern chip





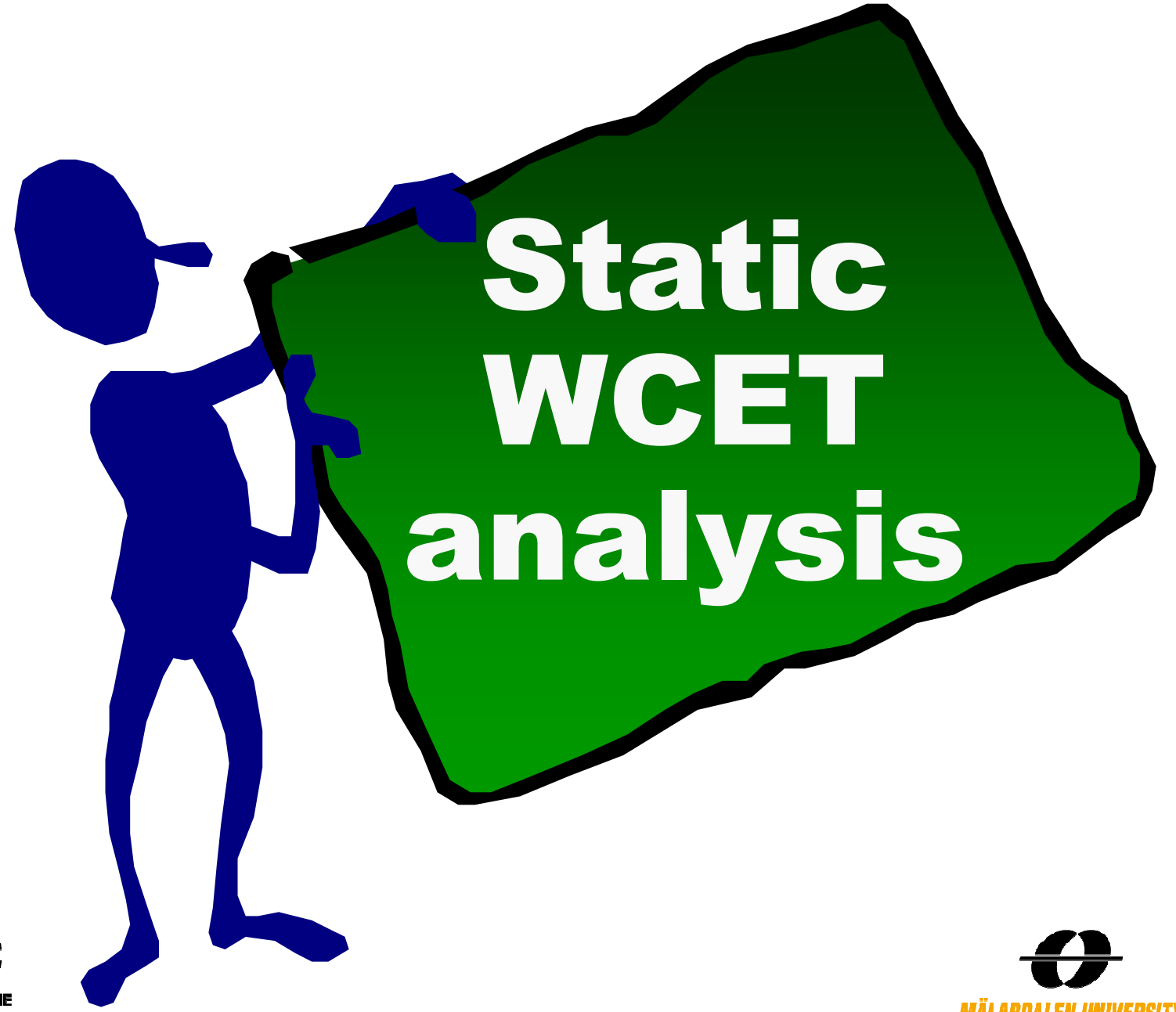
Problem of using measurement

- Measured time never larger than WCET!



A safety margin must be added!

- How much is enough?

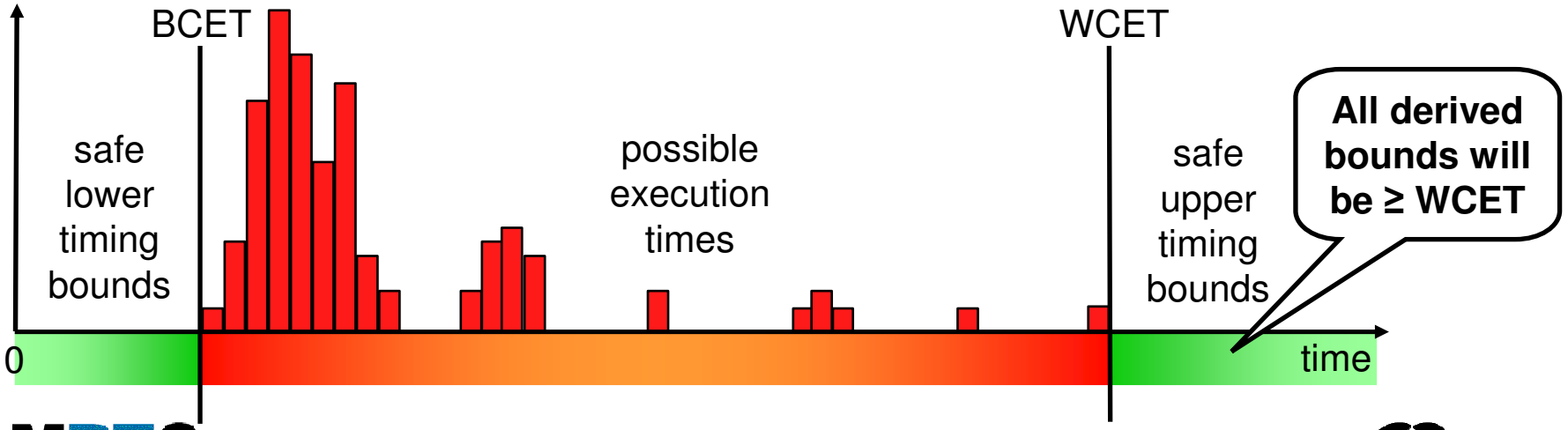


MRTC
MÄLARDALEN REAL-TIME
RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

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**



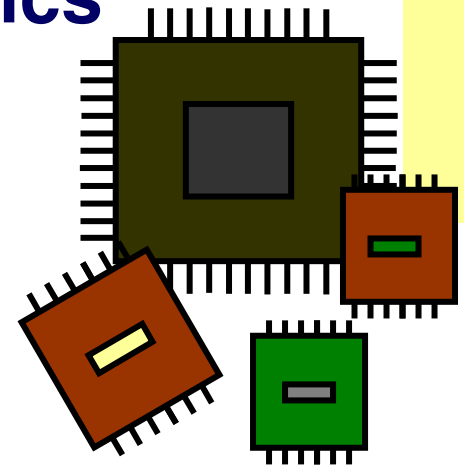
MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

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
  - …

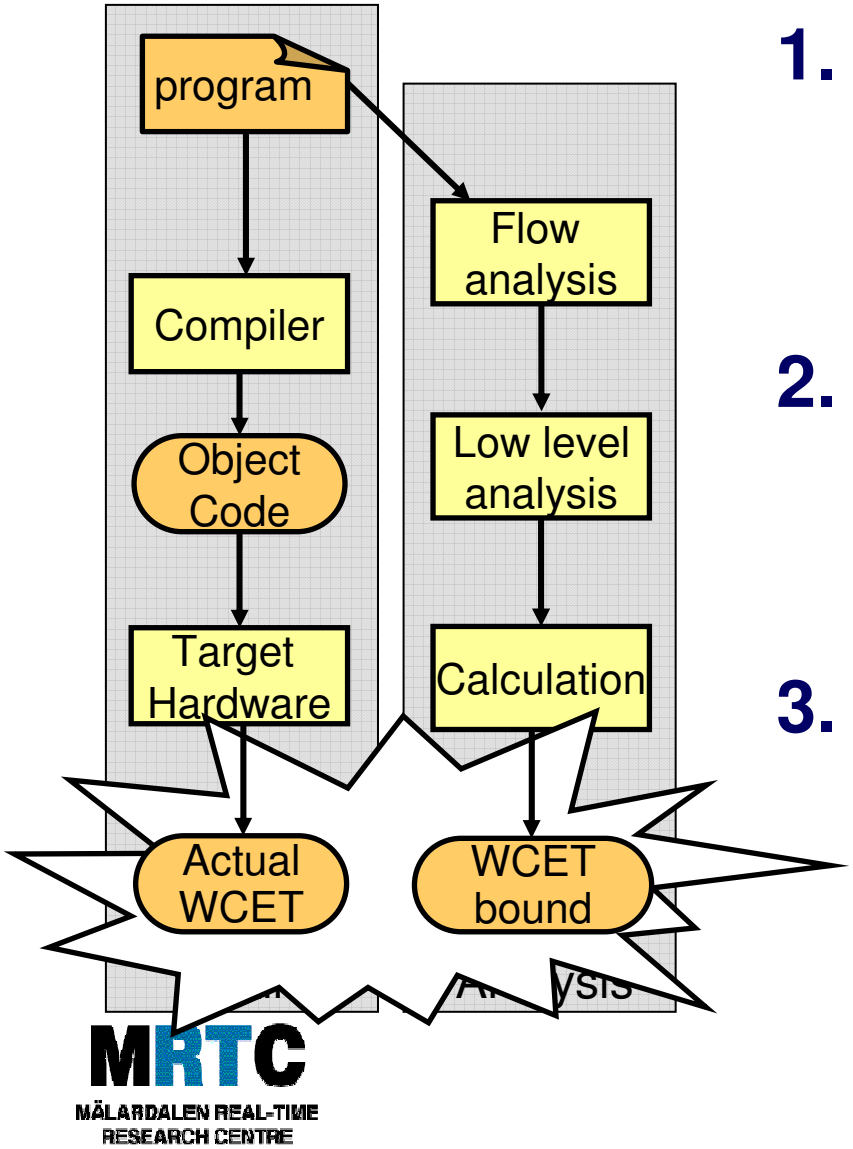
```
foo(x,i):
  while(i < 100)
      if (x > 5) then
        x = x*2;
      else
        x = x+2;
      end
      if (x < 0) then
        b[i] = a[i];
      end
      i = i+1;
end
```



MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

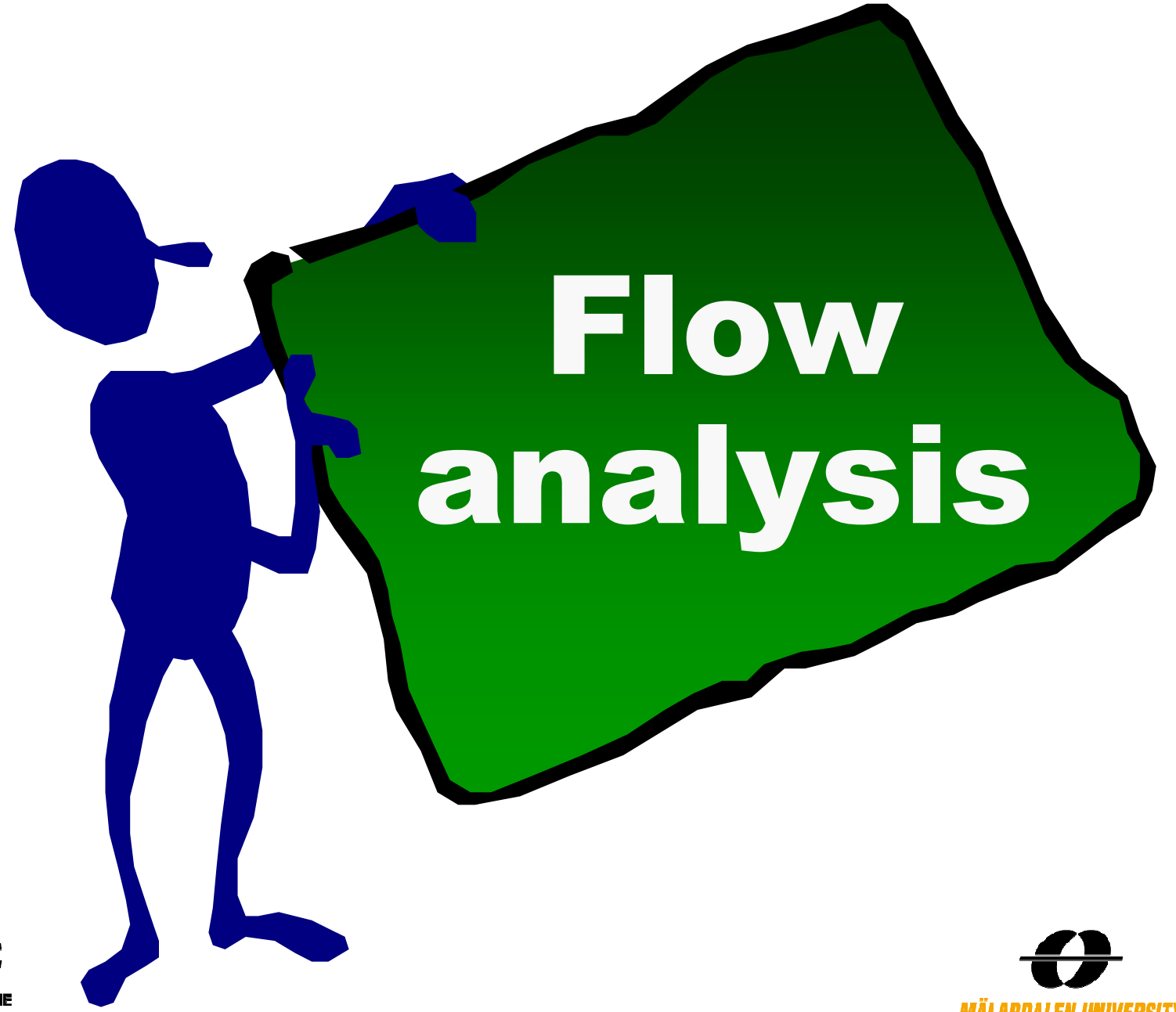
WCET analysis phases



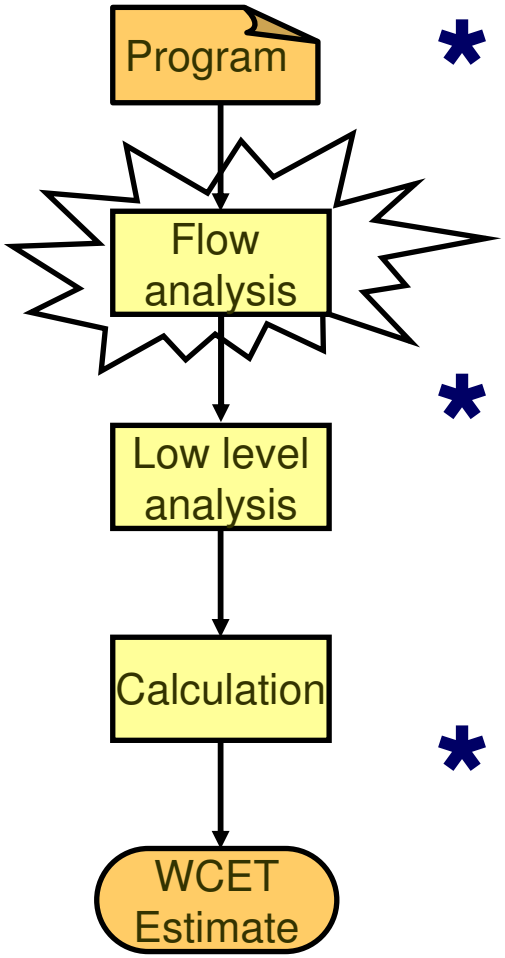
1. **Flow analysis**
   - Bound the number of times different program parts may be executed (mostly SW analysis)
2. **Low-level analysis**
   - Bound the execution time of different program parts (combined SW & HW analysis)
3. **Calculation**
   - Combine flow- and low-level analysis results to derive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

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY


Flow
analysis

Flow Analysis

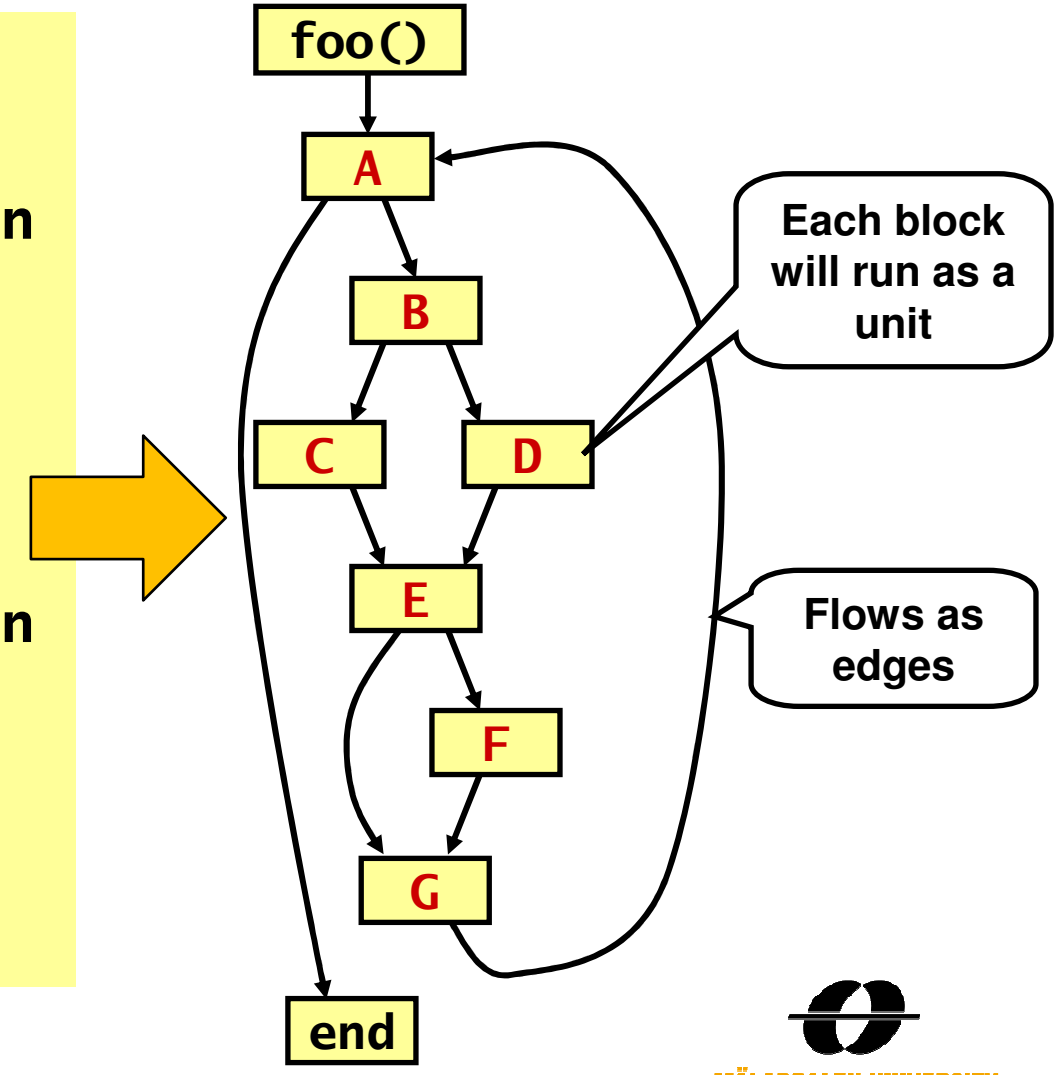


- Provides bounds on the number of times different program parts may be executed
  - Valid for all possible executions
- Examples of provided info:
  - Bounds of loop iterations
  - Bounds on recursion depth
  - Infeasible paths
- Info provided by:
  - Static program analysis
  - Manual annotations

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

The control-flow graph

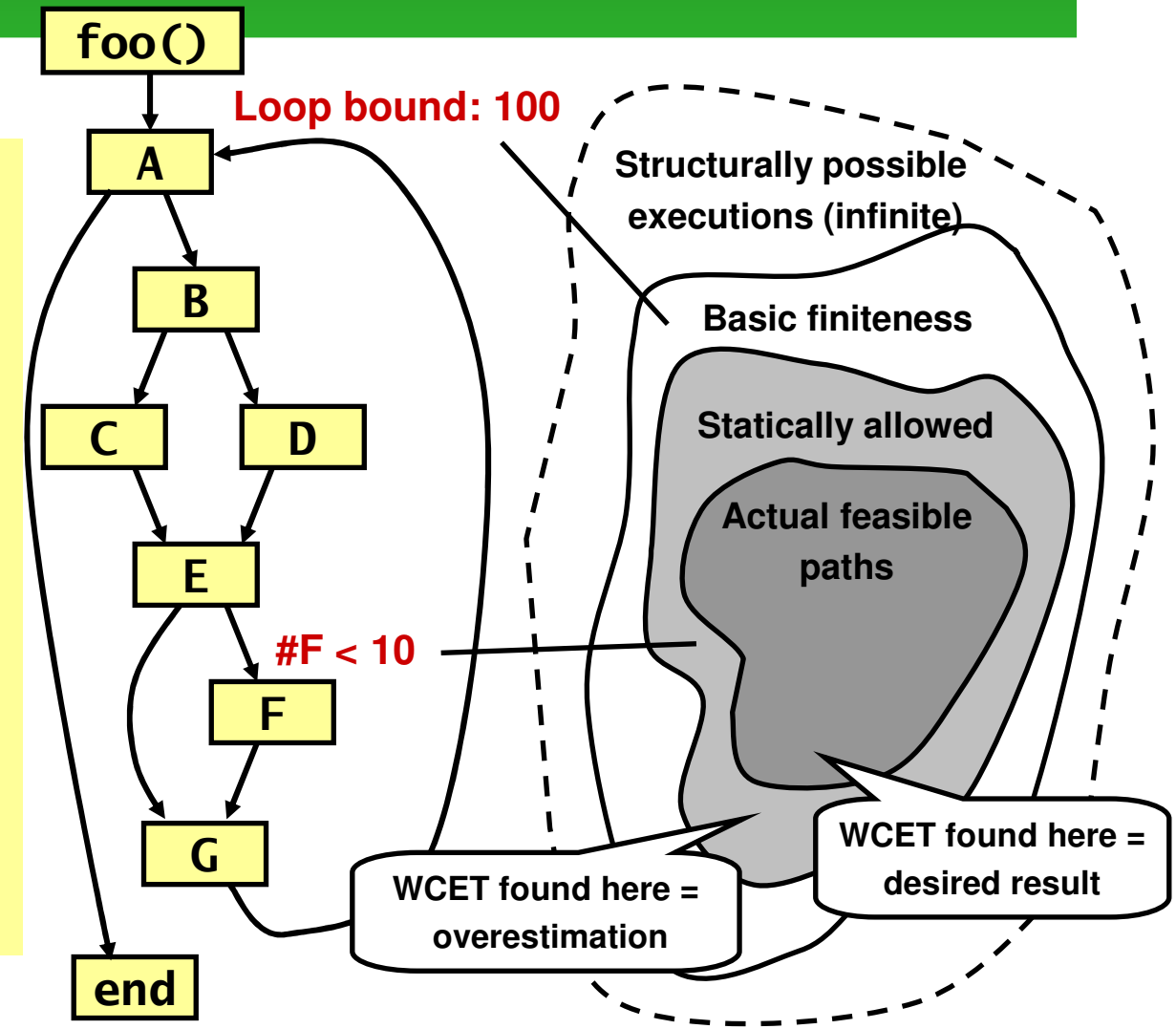
```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```



Flow info characteristics

```
     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 program



Control flow graph

Relation between possible executions and flow info

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

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

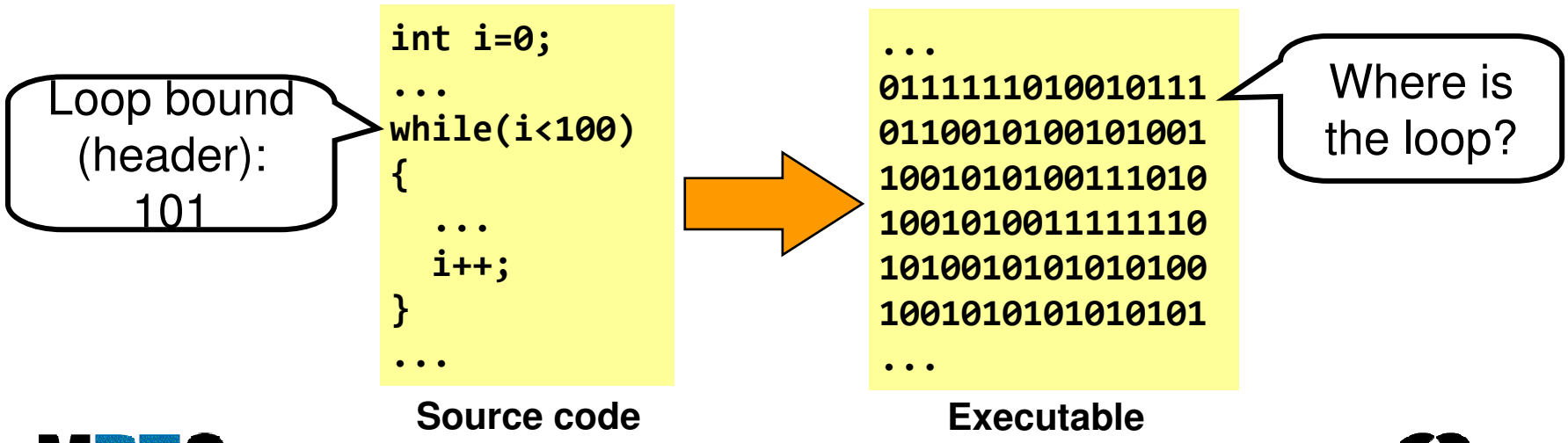
Example: Triangular Loop

```
    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

The mapping problem

- Flow analysis easier on source code level
  - Semantics of code clearer
  - Easier for programmer/tool to derive flow info
- Low-level analysis requires binary code
  - The code executed by the processor
- Question: How to safely map flow source code level flow information to binary code?



The mapping problem (cont)

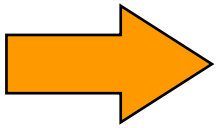
- Embedded compilers often do a lot of code optimizations
  - Important to fit code and data into limited memory resources
- Optimizations may significantly change code (and data) layout
  - After optimizations flow info may no longer be valid
- Solutions:
  - Use special compiler also mapping flow info (not common)
  - Use compiler debug info for mapping (only works with little/no optimizations)
  - Perform flow analysis on binaries (most common)

Flow analysis: Loop condition taken 101 times

Compiler: `i=0` always holds at first execution of loop condition

```
int i=0;
...
while(i<100) {
  ...
  i++;
}
...
```

Before optimization



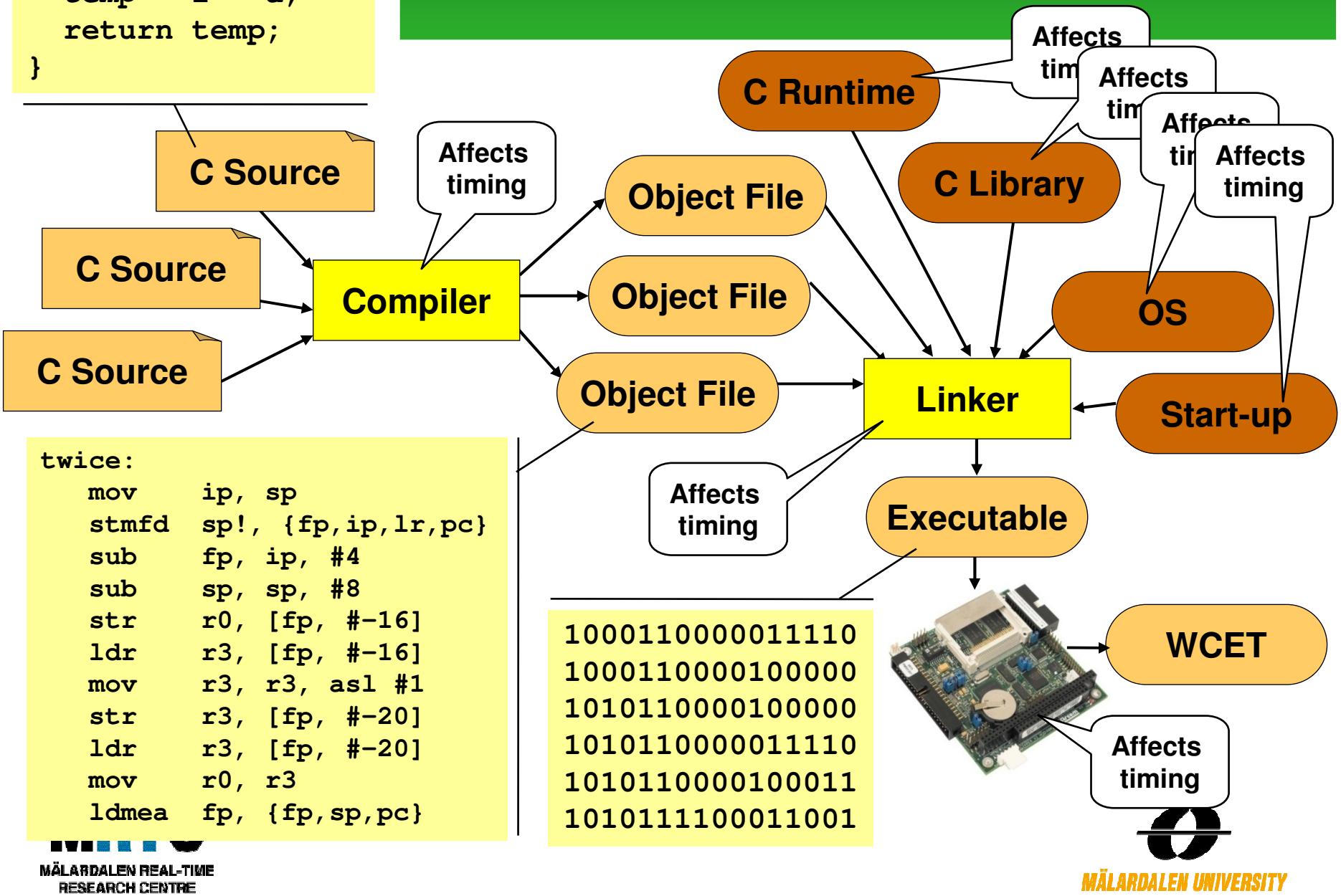
```
int i=0;
...
do {
  ...
  i++;
} while(i<100)
...
```

Loop condition taken 100 times

After optimization

Embedded SW Tool Chain

```
int twice(int a) {
  int temp;
  temp = 2 * a;
  return temp;
}
```




```
twice:
    mov     ip, sp
    stmfd   sp!, {fp,ip,lr,pc}
    sub     fp, ip, #4
    sub     sp, sp, #8
    str     r0, [fp, #-16]
    ldr     r3, [fp, #-16]
    mov     r3, r3, asl #1
    str     r3, [fp, #-20]
    ldr     r3, [fp, #-20]
    mov     r0, r3
    ldmea   fp, {fp,sp,pc}
```

```
1000110000011110
1000110000100000
1010110000100000
1010110000011110
1010110000100011
1010111100011001
```

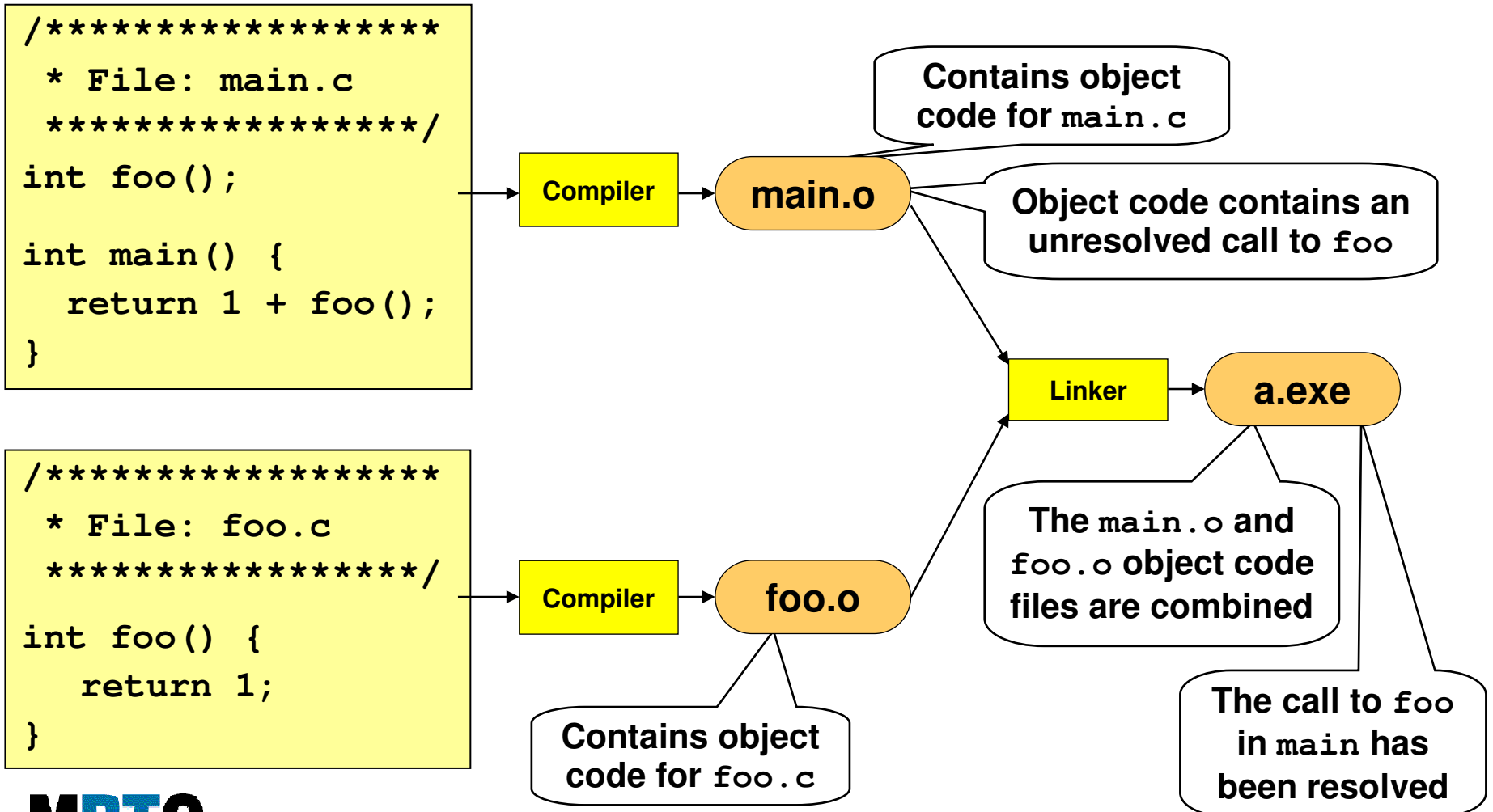
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

The SW building tools

- **The compiler:**
  - Translates an source code file to an object code file
    - Only translates one source code file at the time
  - Often makes some type of code optimizations
    - Increase execution speed, reduce memory size, …
    - Different optimizations give different object code layouts
- **The linker:**
  - Combines several object code files into one executable
    - Places code, global data, stack, etc in different memory parts
    - Resolves function calls and jumps between object files
  - Can also perform some code transformations
- **Both tools may affect the program timing!**

Example: compiling & linking

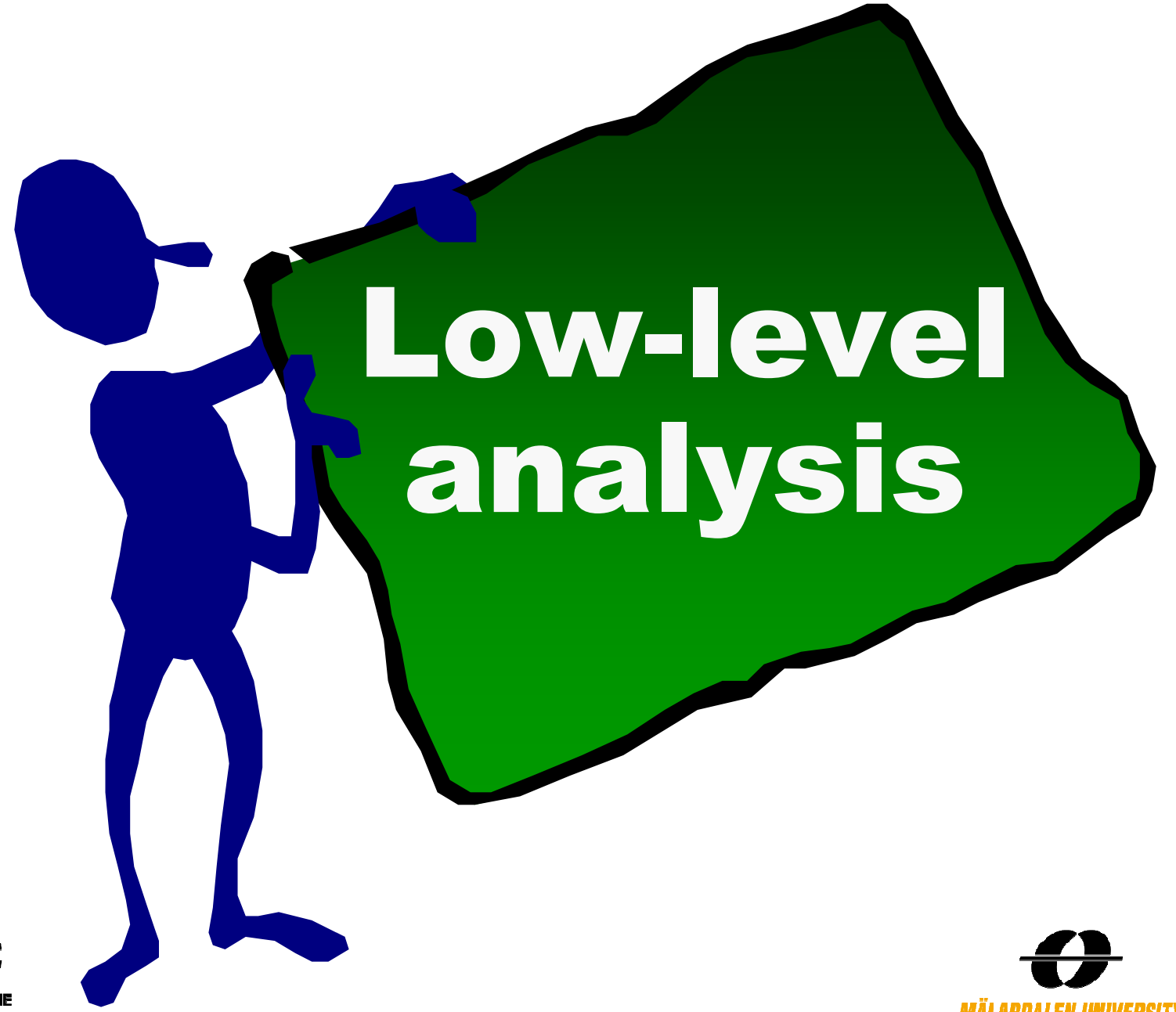


Common additional files

- C Runtime code:
  - Whatever needed but not supported by the HW
    - 32-bit arithmetic on a 16-bit machine
    - Floating-point arithmetic
    - Complex operations (e.g., modulo, variable-length shifts)
  - Comes with the compiler
  - May have a large footprint
    - Bigger for simpler machines
    - Tens of bytes of data and tens of kilobytes of code
- OS code:
  - In many ES the OS code is linked together with the rest of the object code files to form a single binary image

Common additional files

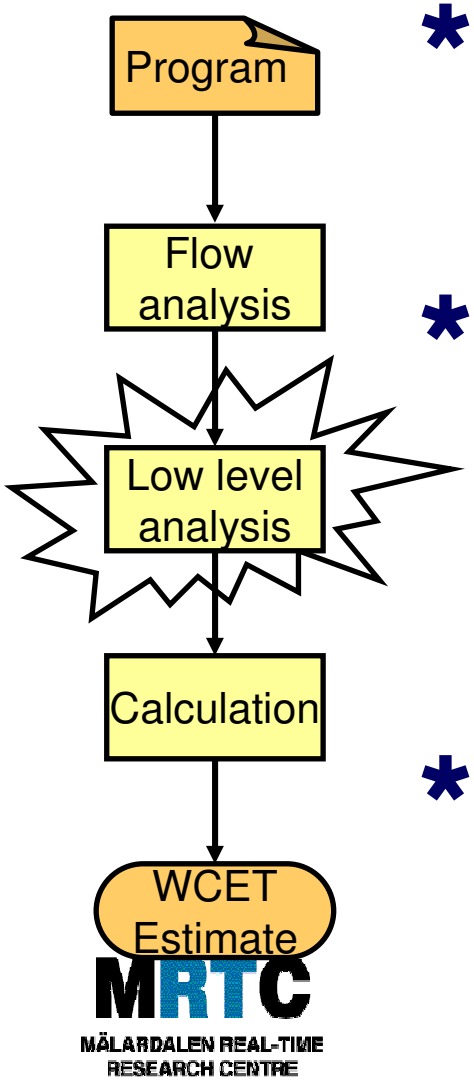
- **Startup code:**
  - A small piece of assembly code that prepares the way for the execution of software written in a high-level language
    - For example, setting up the system stack
  - Many ES compilers provide a file named `startup.asm`, `crt0.s`, … holding startup code
- **C Library code:**
  - A full ANSI-C compiler must provide code that implements all ANSI-C functionality
    - E.g., functions such as `printf`, `memmove`, `strcpy`
  - Many ES compilers only support subset of ANSI-C
  - Comes with the compiler (often non-standard)



MRTC
MÄLARDALEN REAL-TIME
RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Low-Level Analysis



- **Determine execution time bounds for program parts**
  - **Focus of most WCET-related research**
- **Using a model 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**

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Some HW model details

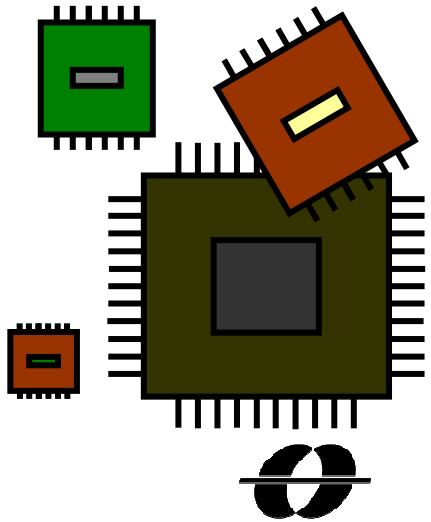
- Much effort required to safely model CPU internals
  - Pipelines, branch predictors, superscalar, out-of-order, …
- Much effort to safely model memories
  - Cache memories must be modelled in detail
  - Other types of memories may also affect timing
- For complex CPUs many features must be analyzed together
  - Timing of instructions get very *history dependant*
- Developing a safe HW timing model troublesome
  - May take many months (or even years)
  - All things affecting timing must be accounted for

Hardware time variability

- Simpler 4-, 8- & 16-bit processors (H8300, 8051, …):
  - Instructions might have varying execution time due to argument values
  - Varying data access time due to different memory areas
  - Analysis rather simple, timing fetched from HW manual
- Simpler 16- & 32-bit processors, with a (scalar) pipe-line and maybe a cache (ARM7, ARM9, V850E, …):
  - Instruction timing dependent on previously executed instructions and accessed data:
    - ➡State of pipeline and cache
  - Varying access times due to cache hits and misses
  - Varying pipeline overlap between instructions
  - Hardware features can be analyzed in isolation

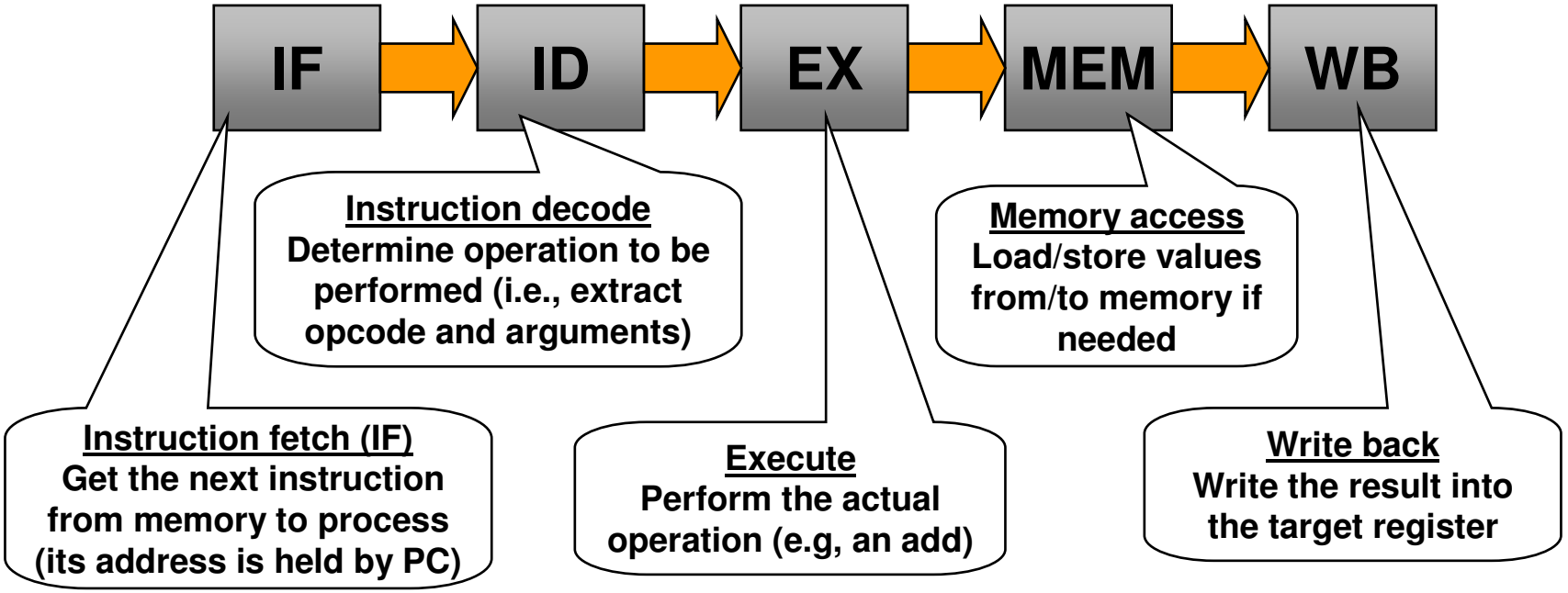
Hardware time variability

- **Advanced 32- & 64-bit processors (PowerPC 7xx, Pentium, UltraSPARC, ARM11, …):**
  - Many performance enhancing features affect timing
    - Pipelines, out-of-order exec, branch pred., caches, speculative exec.
    - Instruction timing gets very history dependent
  - Some processors suffer from *timing anomalies*
    - E.g., a cache miss might give shorter overall program execution time than a cache hit
  - Features and their timing interact
    - Most features must be analyzed together
  - Hard to create a correct and safe hardware timing model!
- **Multi-cores - discussed later**



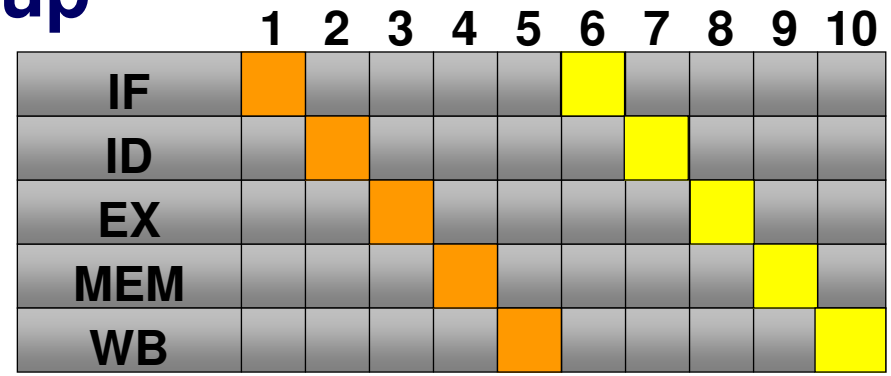
Example: CPU pipelines

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

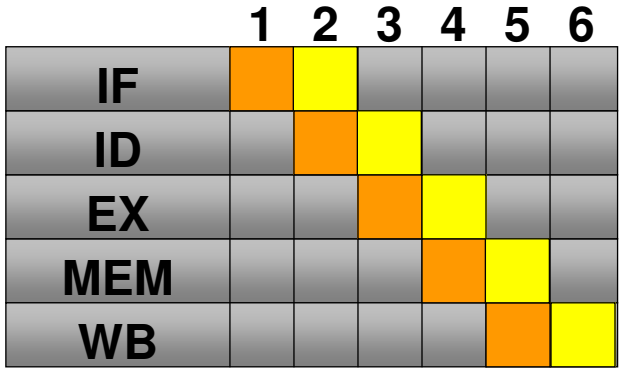


CPU pipelines

- 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



- Pipelining:
  - In principle: speedup = pipeline length
  - However, often dependencies between instructions



Example: RAW dependency

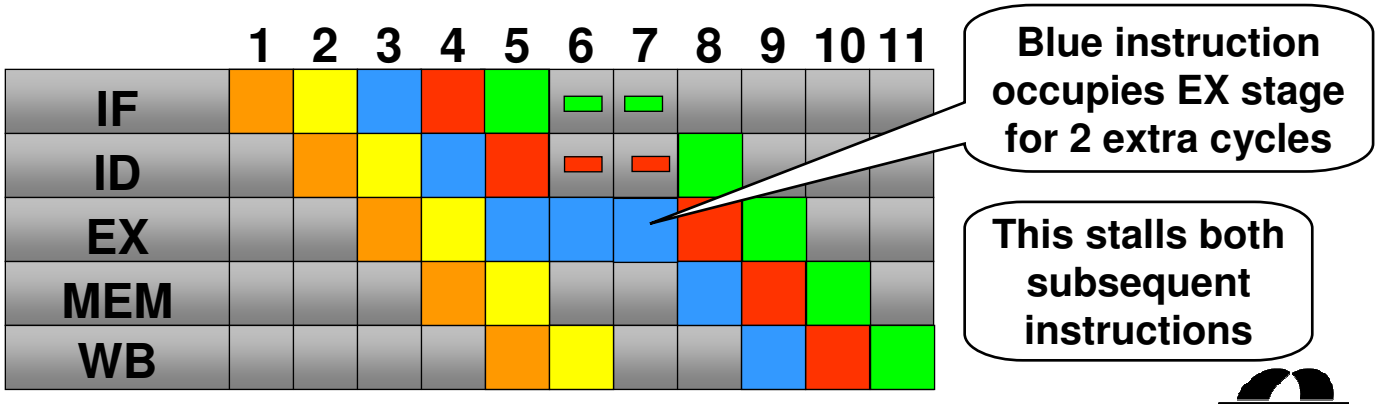
I2 depends on completion of data write of I1

```
I1. add $r0, $r1, $r2
I2. sub $r3, $r0, $r4
```

May cause pipeline stall

Pipeline Variants

- **None:** Simple CPUs (68HC11, 8051, …)
- **Scalar:** Single pipeline (ARM7,ARM9,V850, …)
- **VLIW:** Multiple pipelines, static, compiler scheduled (DSPs, Itanium, Crusoe, …)
- **Superscalar:** Multiple pipelines, out-of-order (PowerPC 7xx, Pentium, UltraSPARC, ...)



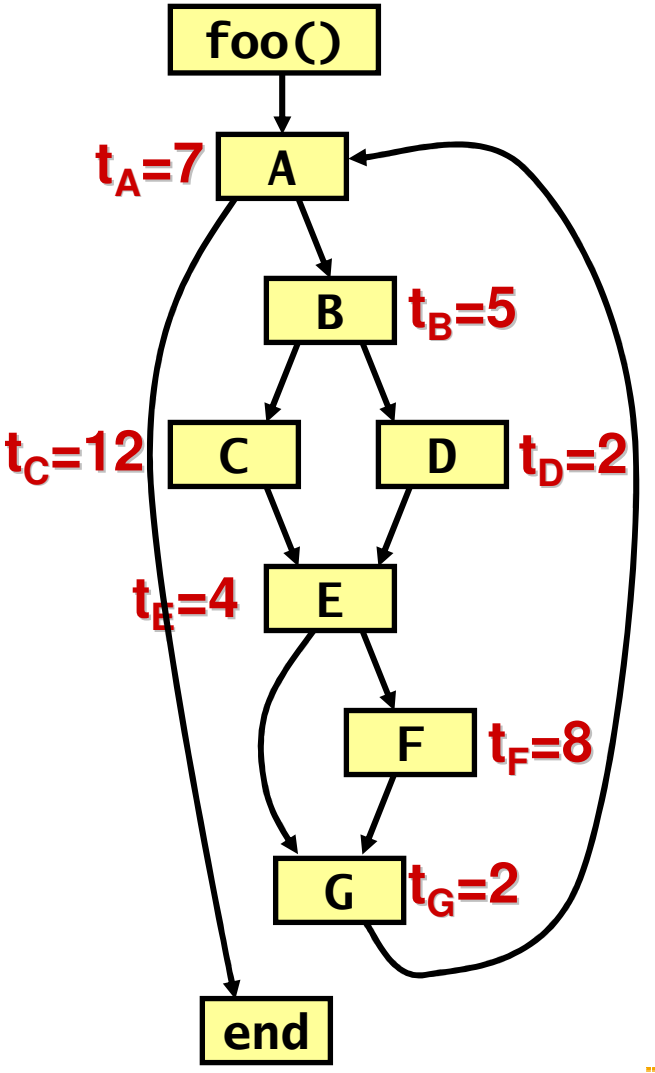
Example: No Pipeline

```
     foo(x,i):
A:    while(i < 100)        (7 cycles)
B:      if (x > 5) then     (5 c)
C:        x = x*2;          (12 c)
        else
D:        x = x+2;          (2 c)
        end
E:      if (x < 0) then     (4 c)
F:        b[i] = a[i];      (8 c)
        end
G:      i = i+1;            (2 c)
      end
```

- Constant time for each block in the code
- Object code not shown

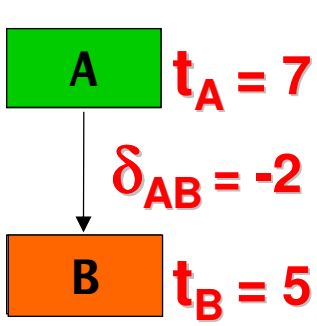
Example: No pipeline

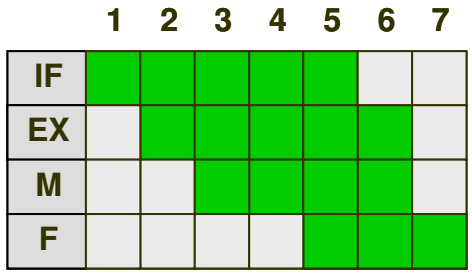
```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```

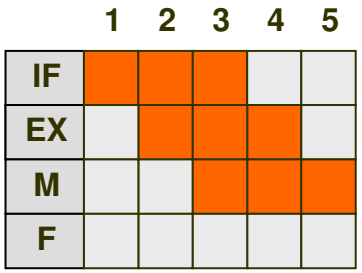


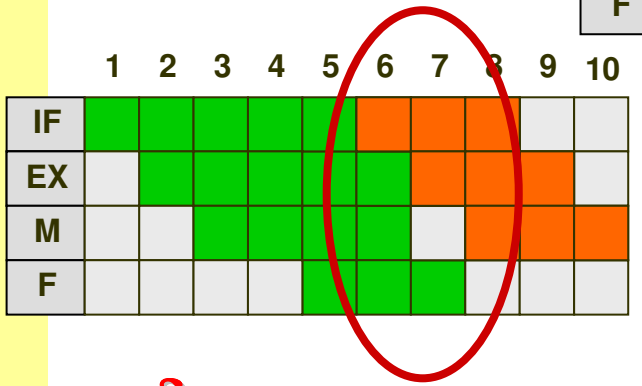
Example: Simple Pipeline

```
     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
```









$t_{AB} = 10$

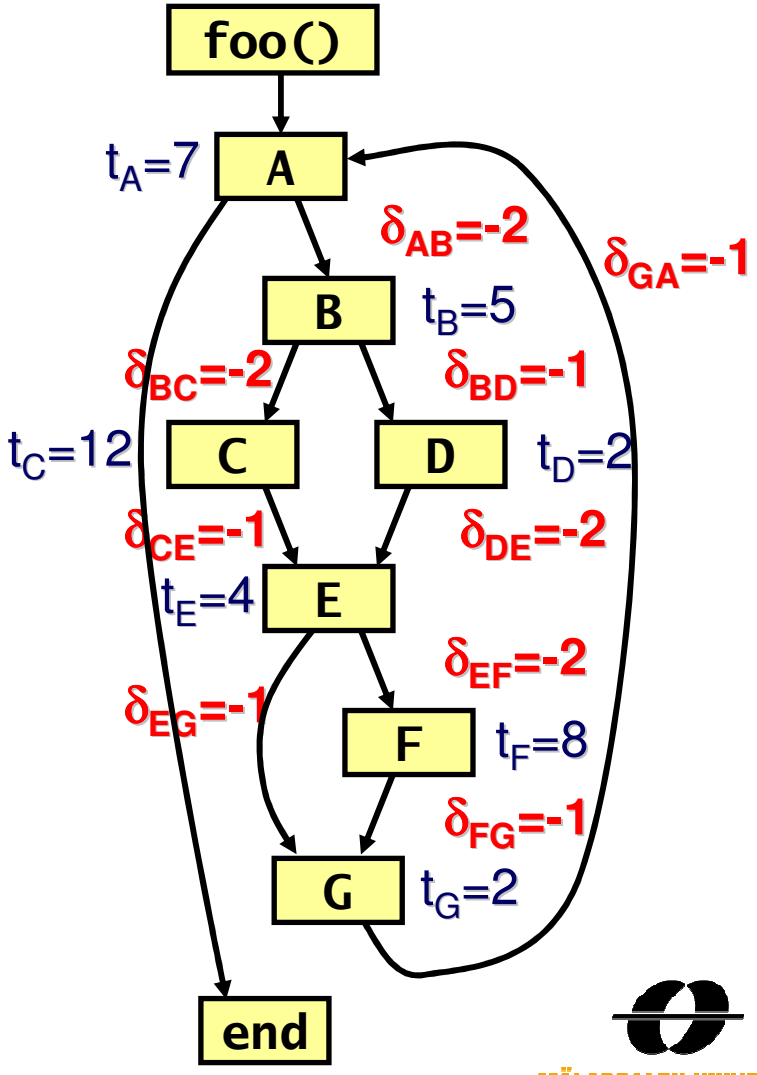
$\delta_{AB} = 10 - (7 + 5) = -2$

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

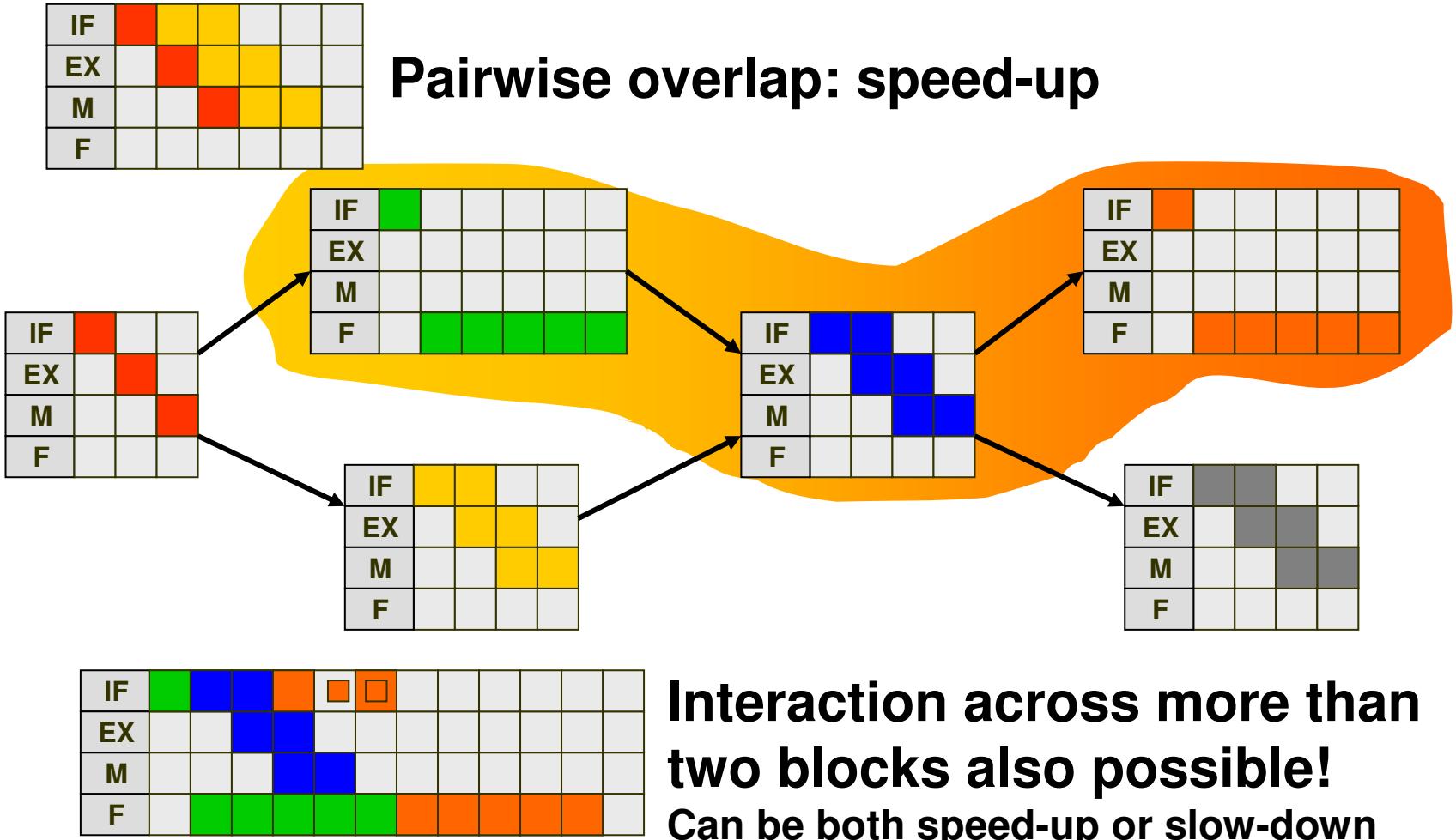
MÄLARDALEN UNIVERSITY

Example: Pipeline result

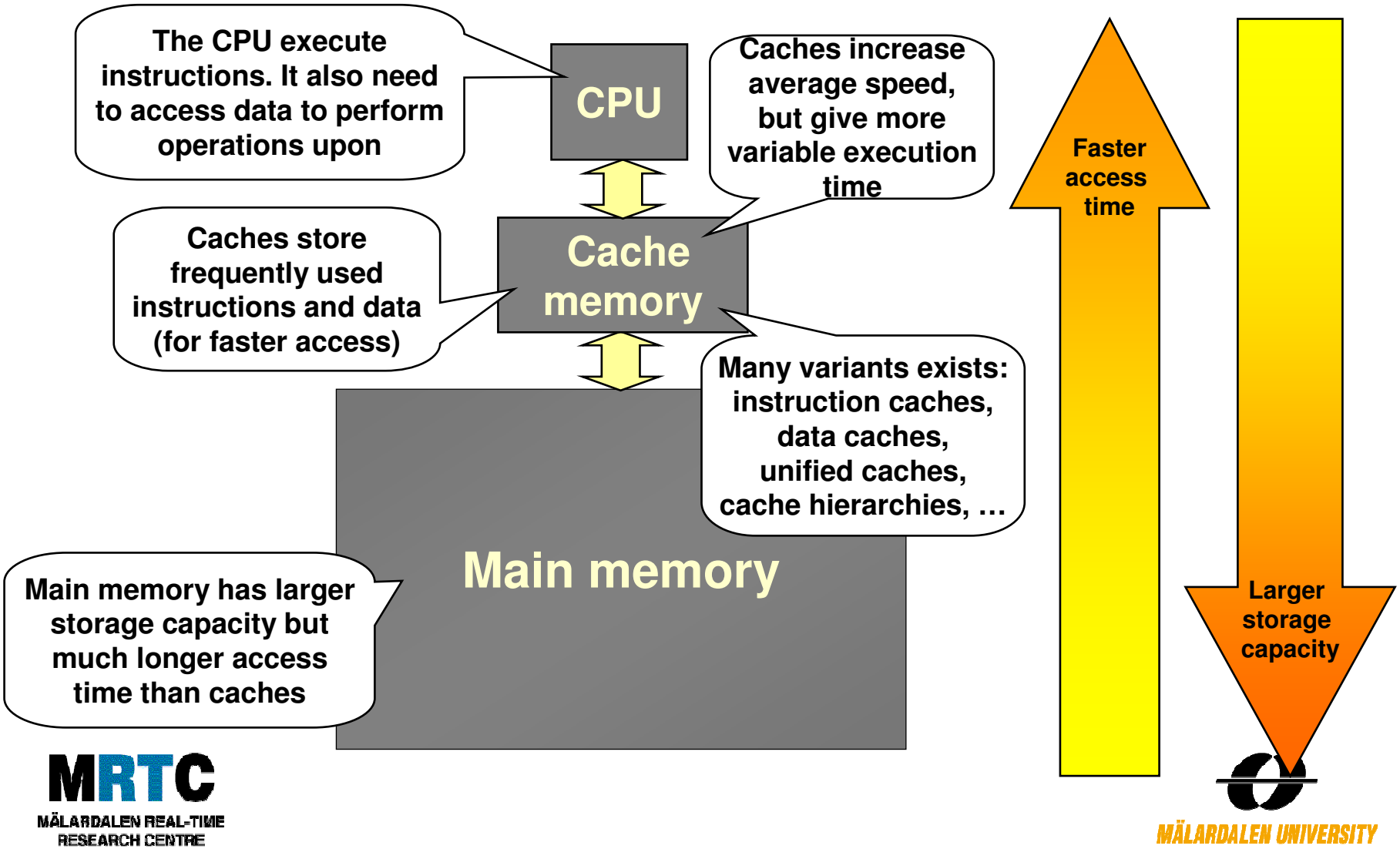
```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```



Pipeline Interactions



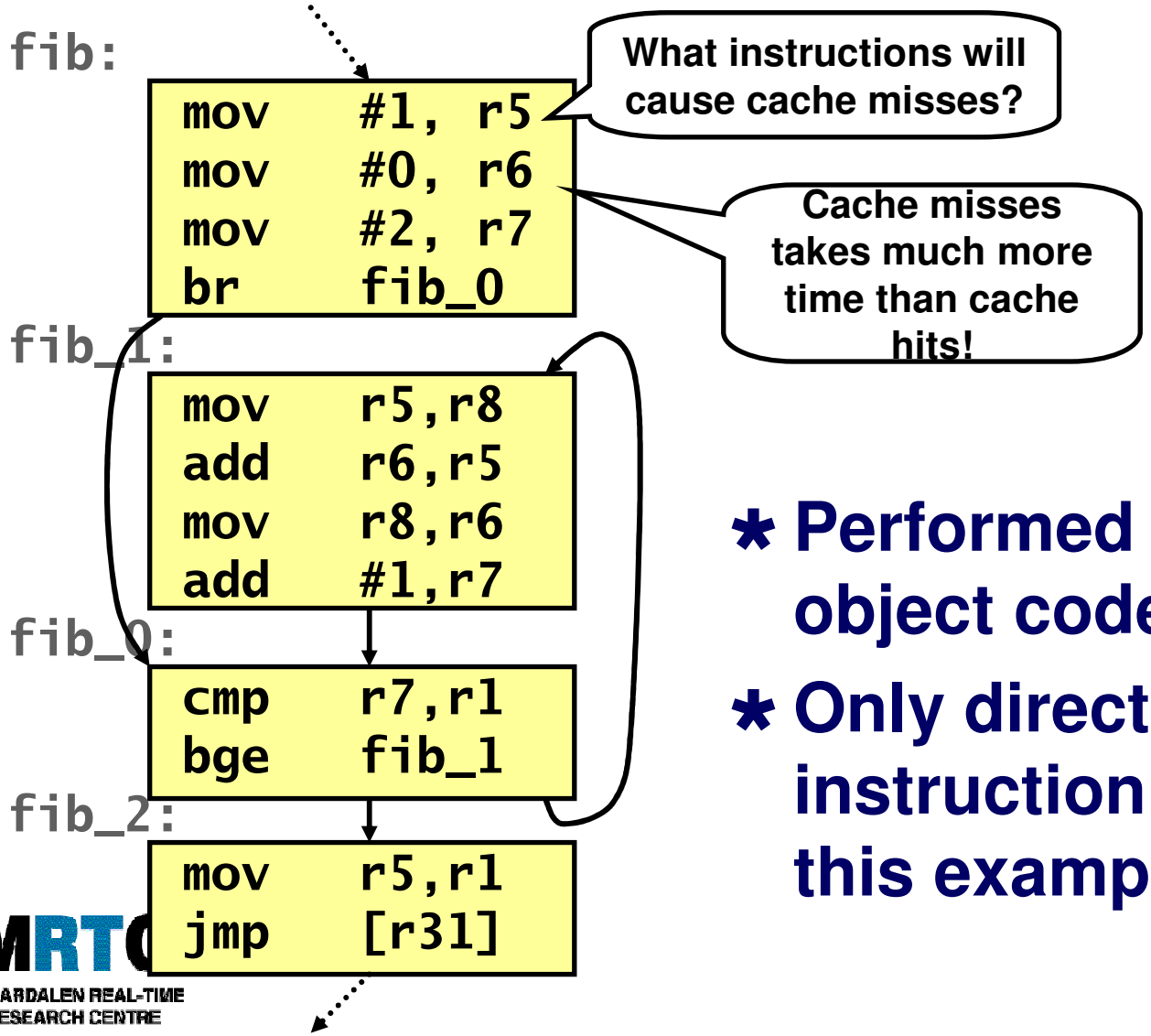
The memory hierarchy

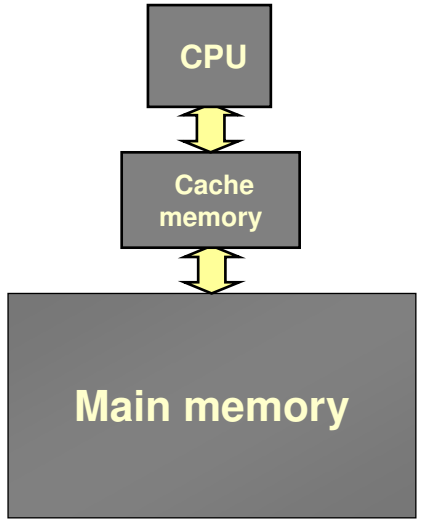


MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Example: Cache analysis

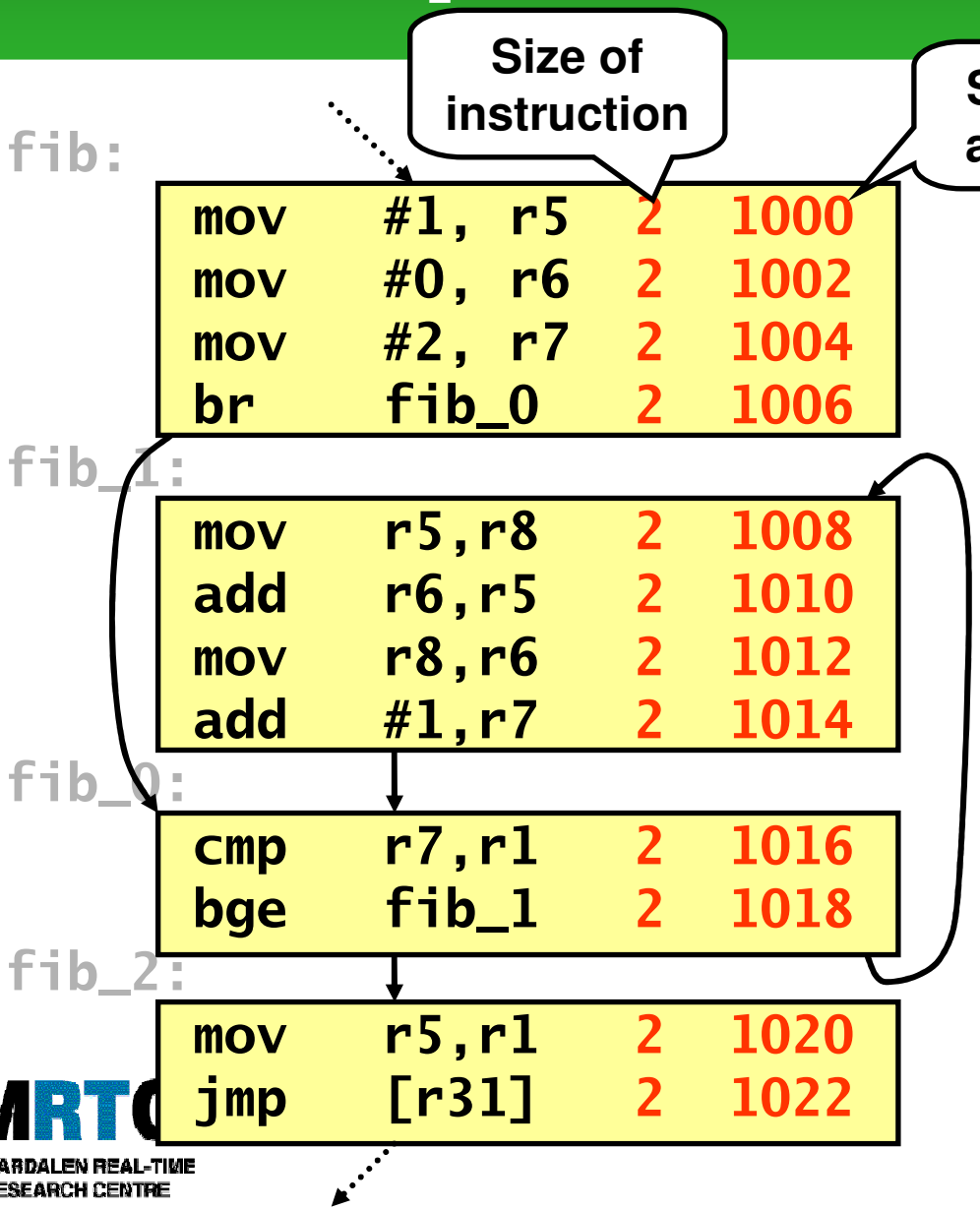




- Performed on the object code
- Only direct-mapped instruction cache in this example

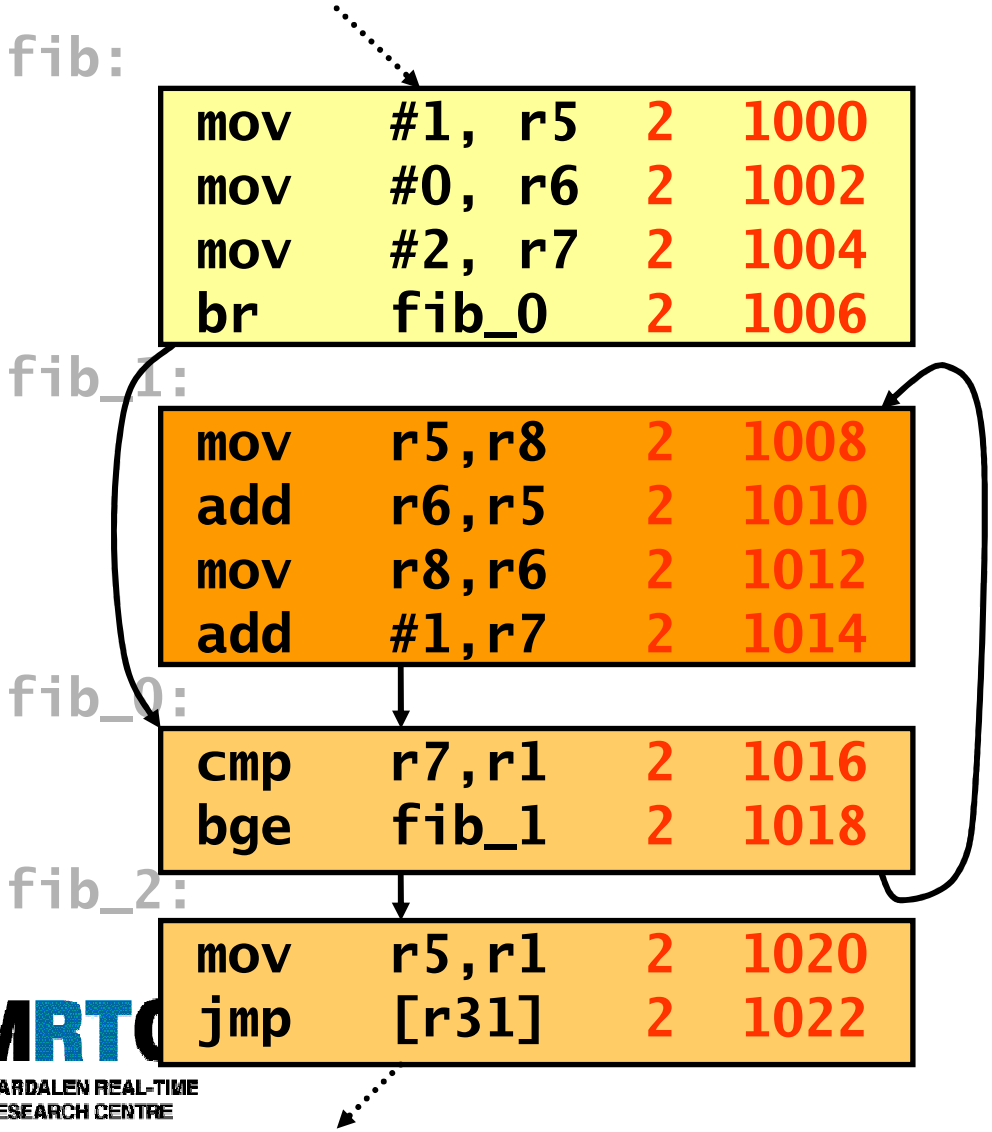
MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

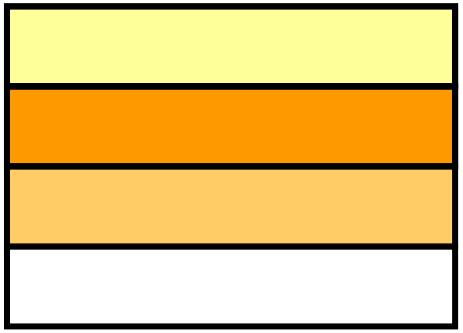
Example: Cache analysis



- Information needed for instruction cache analysis

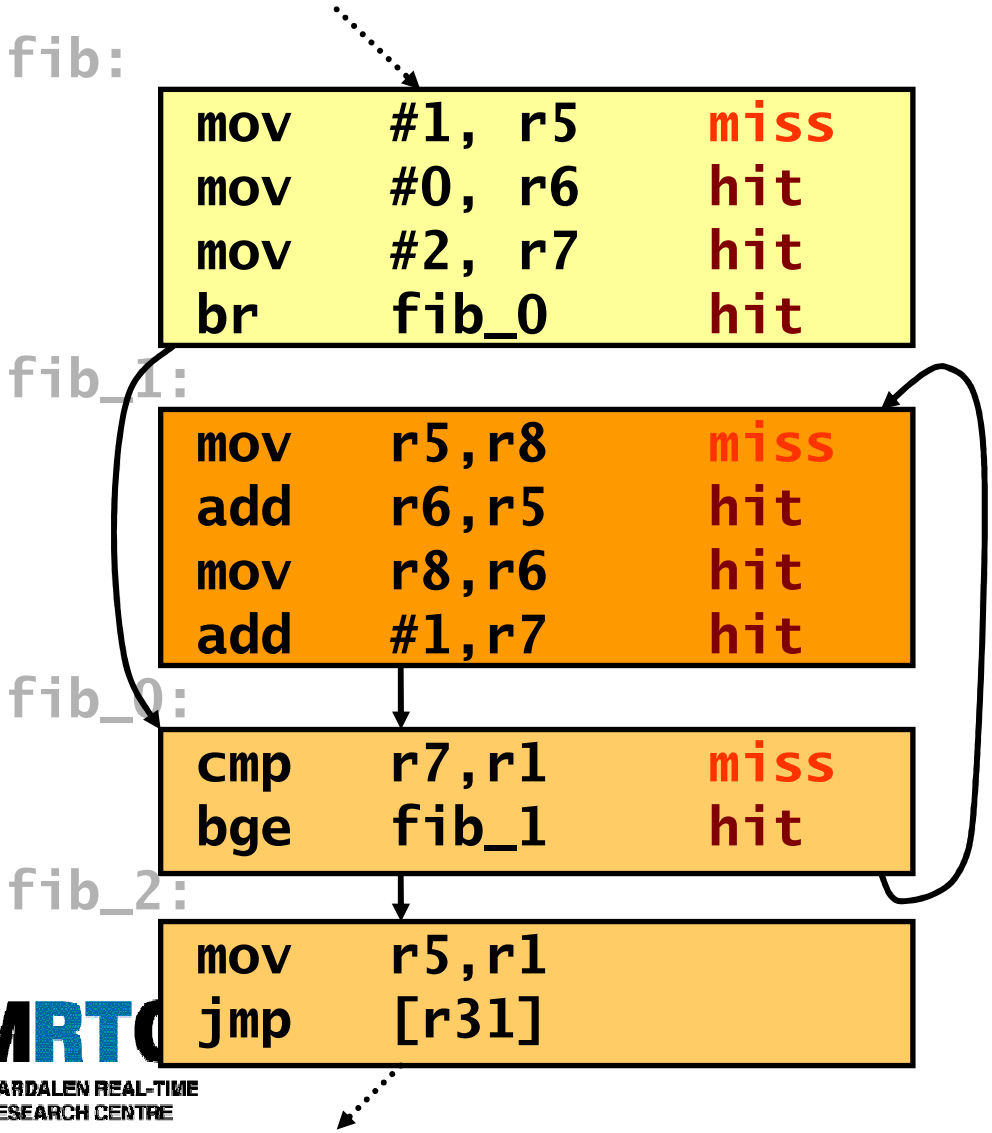
Example: Cache analysis

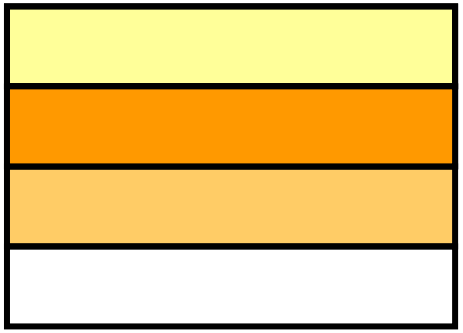




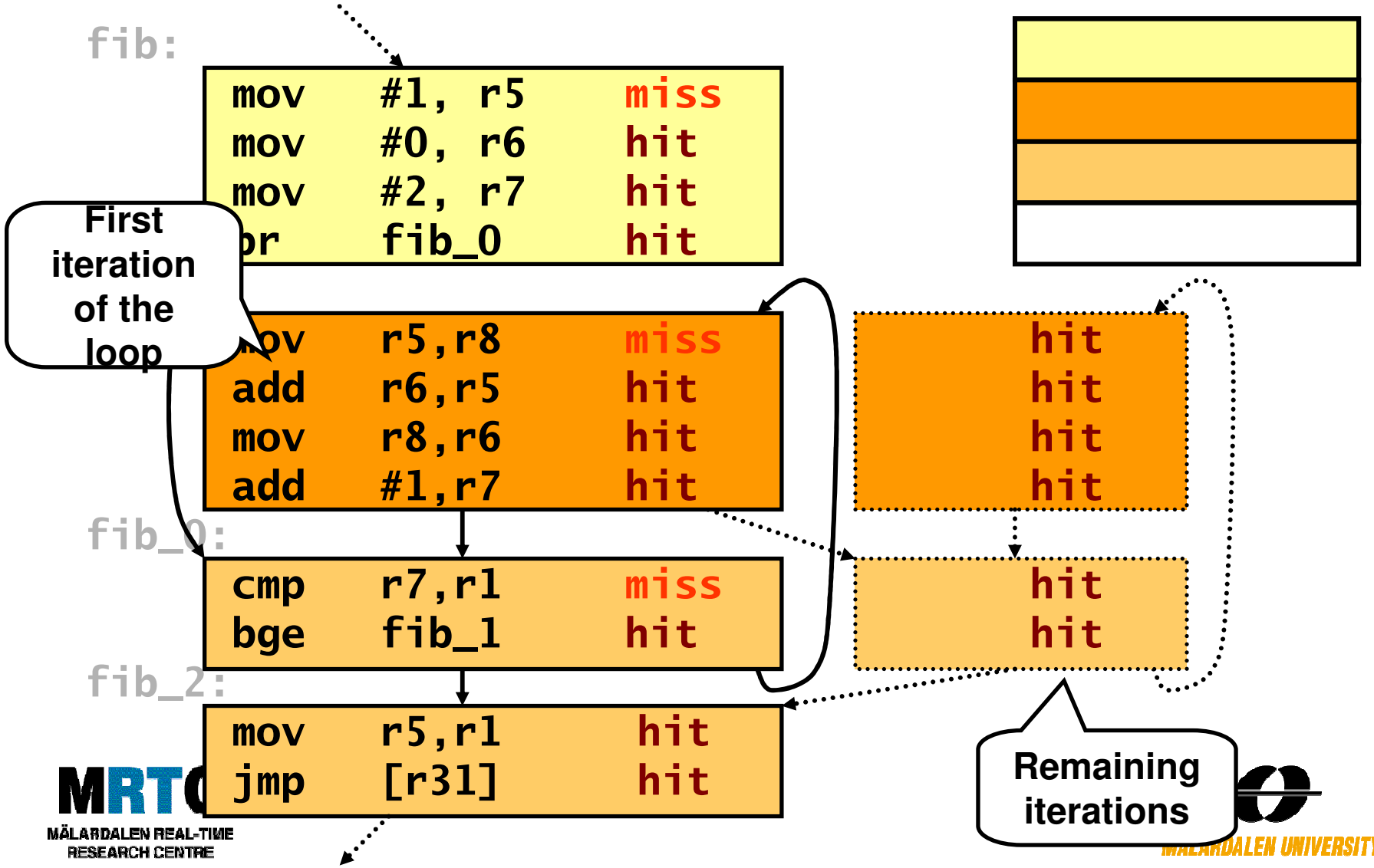
- Mapping to instruction cache

Example: Cache analysis



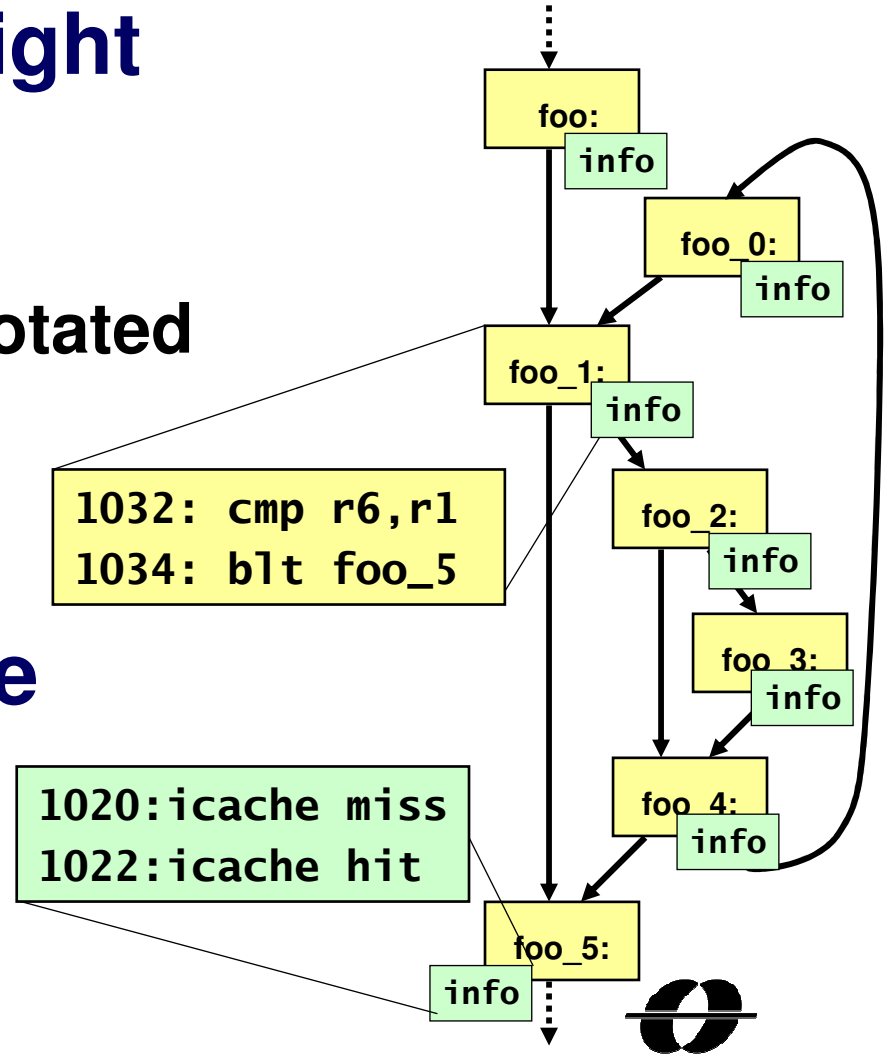


Example: Cache analysis



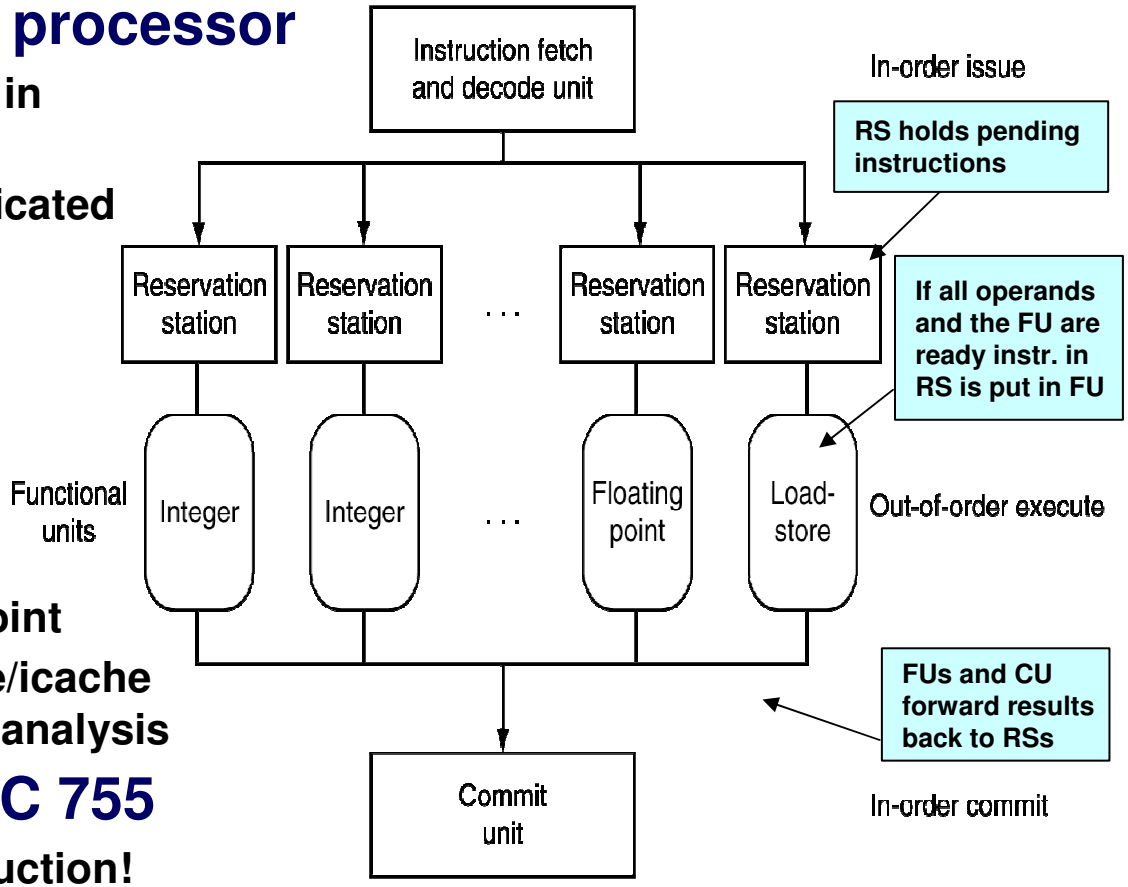
Cache & Pipeline analysis

- 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



Analysis of complex CPUs

- **Example: Out-of-order processor**
  - Instructions may executes in parallel in functional units
  - Functional units often replicated
  - Dynamic scheduling of instructions
  - Do not need to follow issuing order
- **Very difficult analysis**
  - Track all possible pipeline states, iterate until fixed point
  - Require integrated pipeline/icache /dcache/branch-prediction analysis
- **Been done for PowerPC 755**
  - Up to 1000 states per instruction!

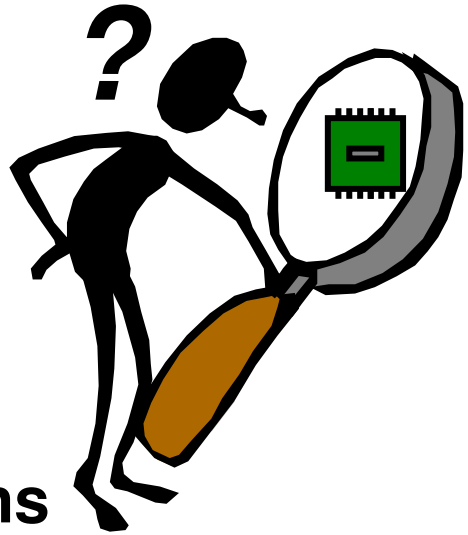


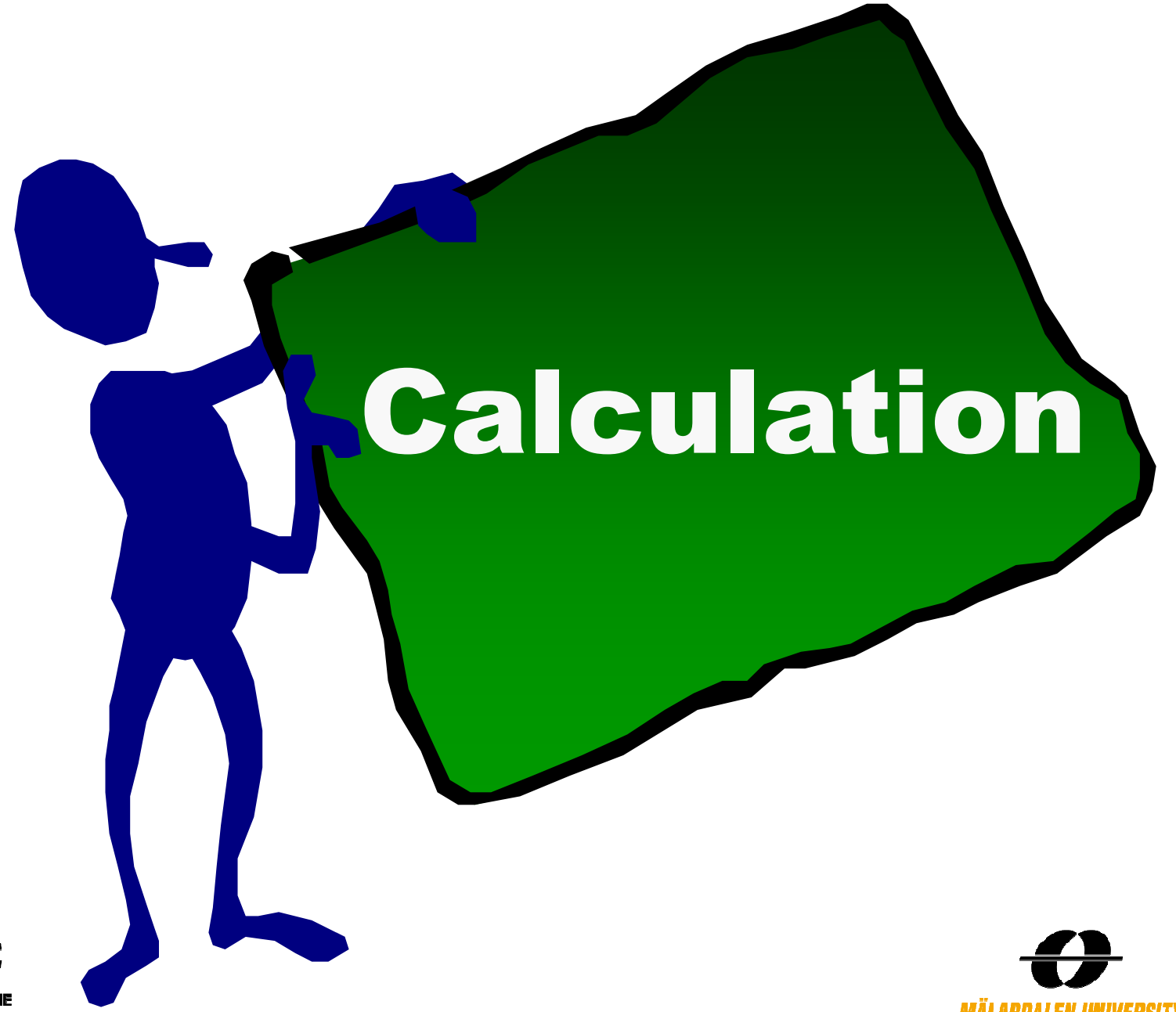
MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Low-level analysis correctness?

- Abstract model of the hardware is used
- Modern hardware often very complex
  - Combines many features
  - Pipelining, caches, branch prediction, out-of-order...
- Have all effects been accounted for?
  - Manufactures keep hardware internals secret
  - Bugs in hardware manuals
  - Bugs relative hardware specifications

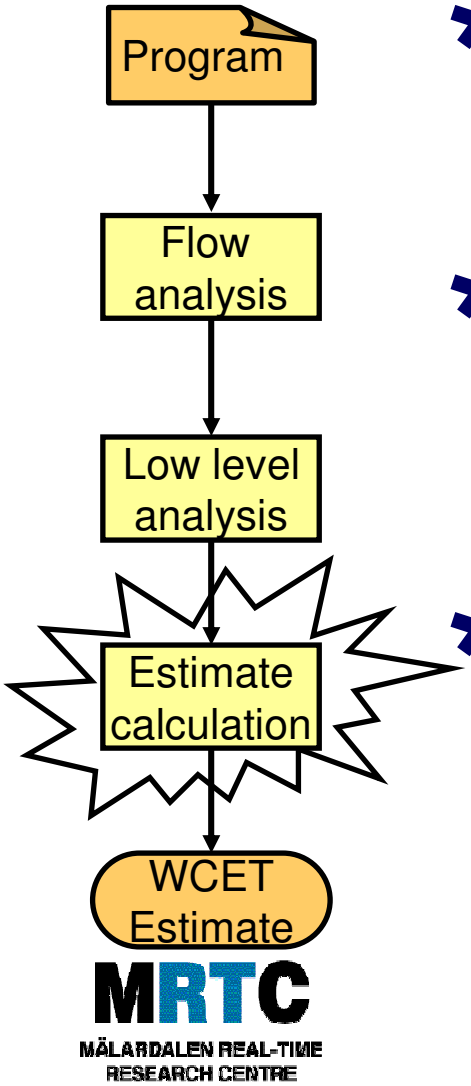




MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Calculation



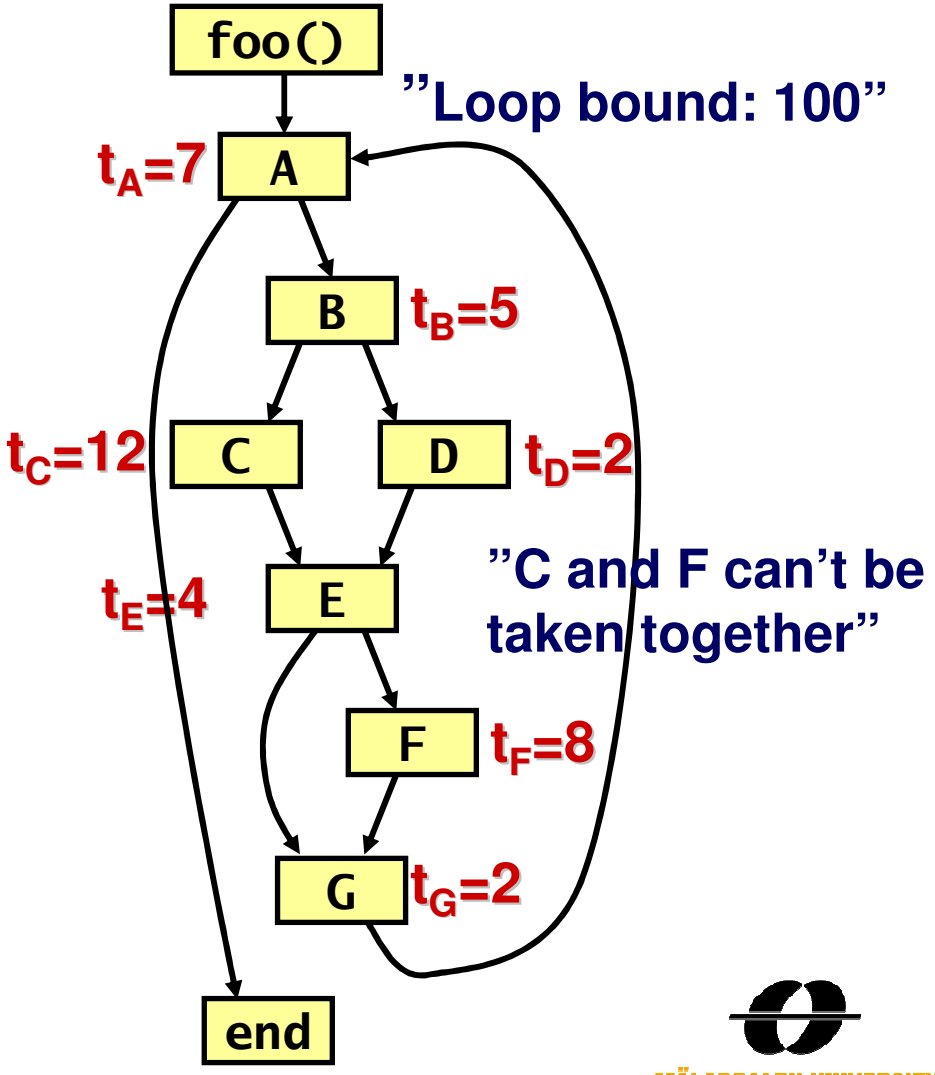
- **Derive an upper bound on the program's WCET**
  - Given flow and timing information
- **Several approaches used:**
  - Tree-based
  - Path-based
  - Constraint-based (IPET)
- **Properties of approaches:**
  - Flow information handled
  - Object code structure allowed
  - Modeling of hardware timing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

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

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
```



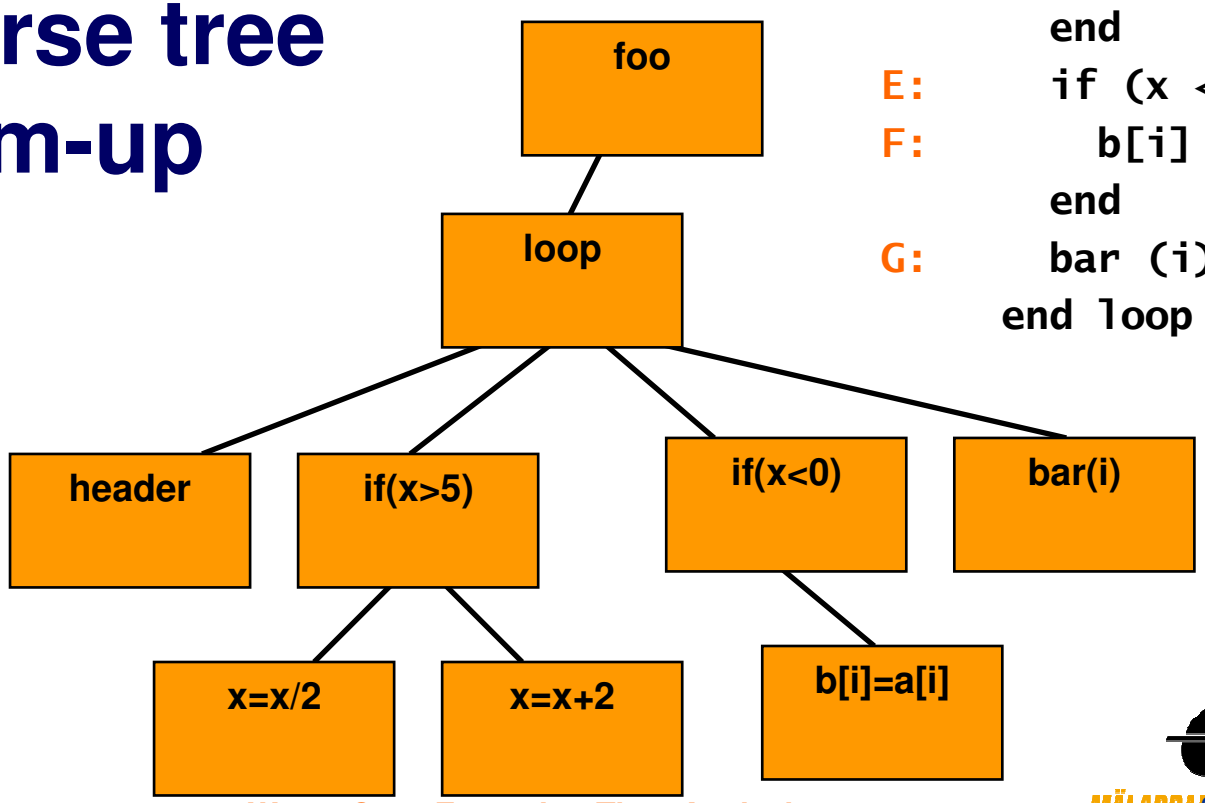
MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Tree-Based Calculation

- Use syntax-tree of program
- Traverse tree bottom-up

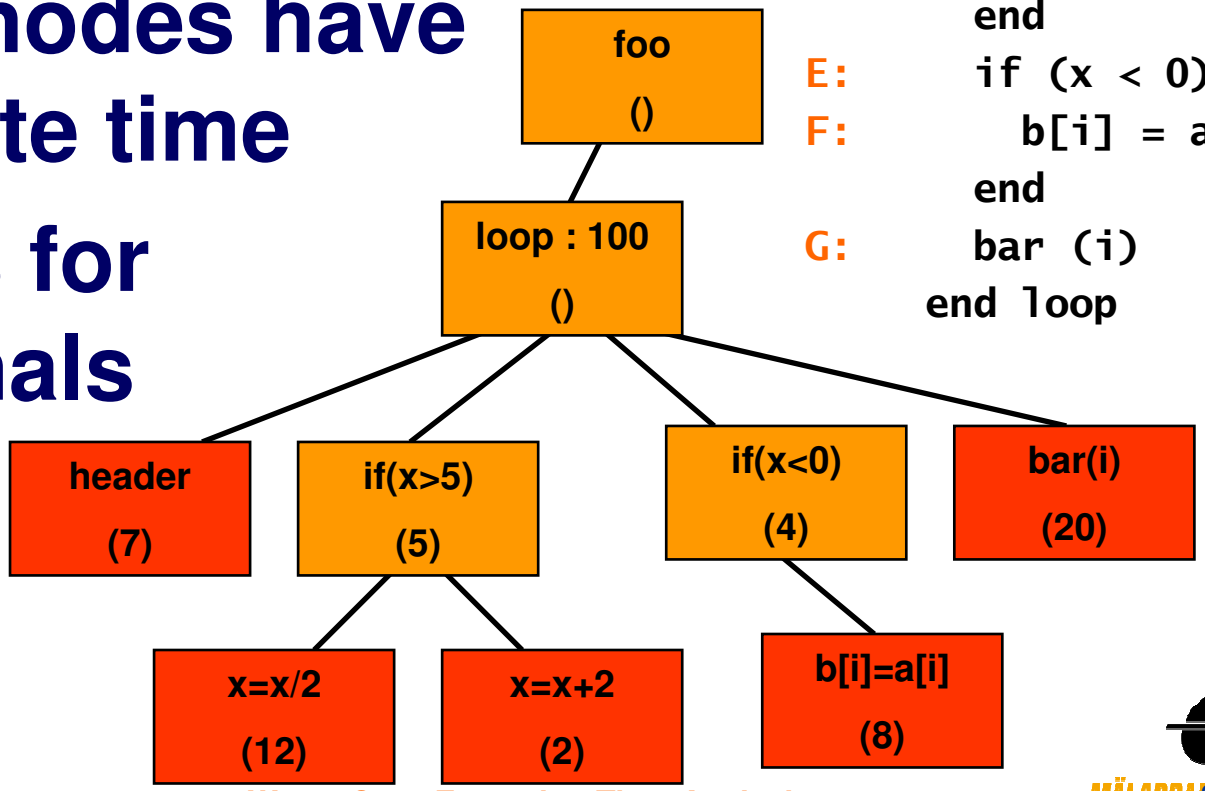
```
     foo(x):
A:    loop(i=1..100)
B:      if (x > 5) then
C:        x = x*2
        else
D:        x = x+2
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      bar (i)
      end loop
```



Tree-Based Calculation

- Use constant time for nodes
- Leaf nodes have definite time
- Rules for internals

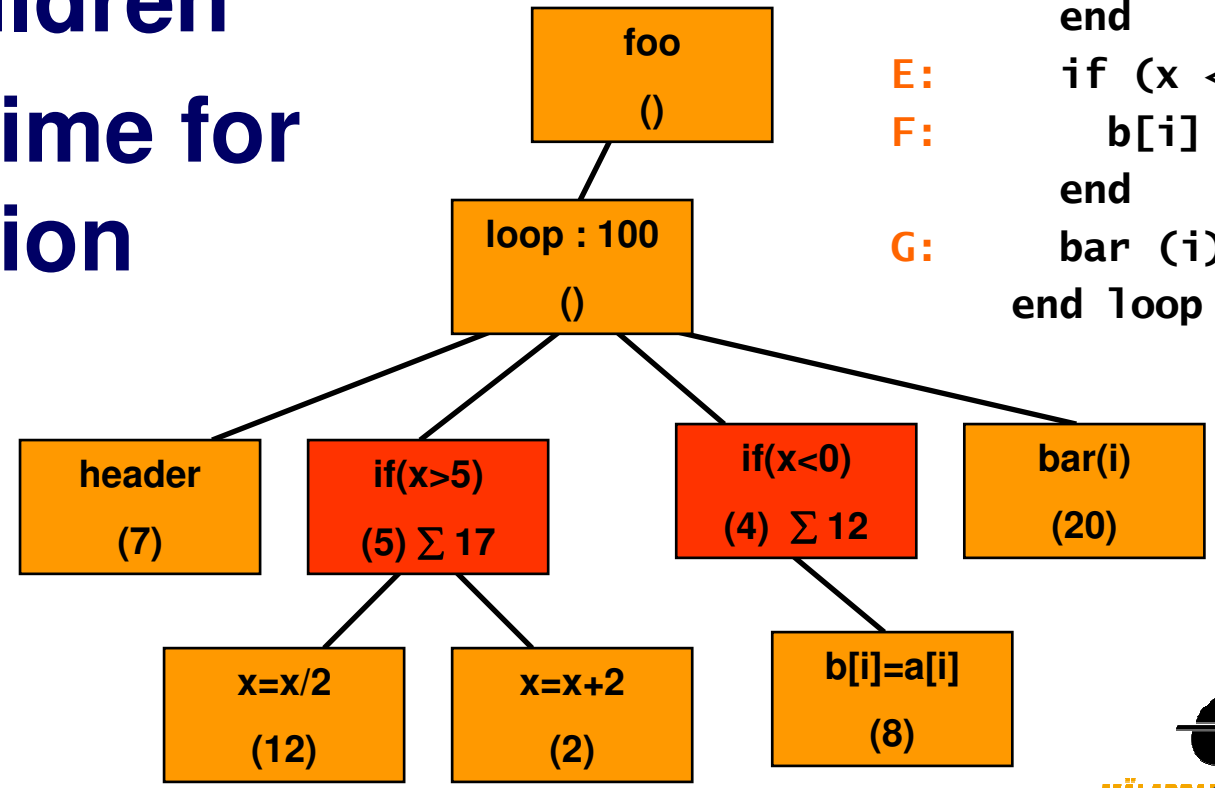
```
     foo(x):
A:    loop(i=1..100)      (7 c)
B:      if (x > 5) then   (5 c)
C:        x = x*2         (12 c)
        else
D:        x = x+2         (2 c)
        end
E:      if (x < 0) then   (4 c)
F:        b[i] = a[i];    (8 c)
        end
G:      bar (i)           (20 c)
      end loop
```



Tree-Based: IF statement

- For a decision statement: max of children
- Add time for decision itself

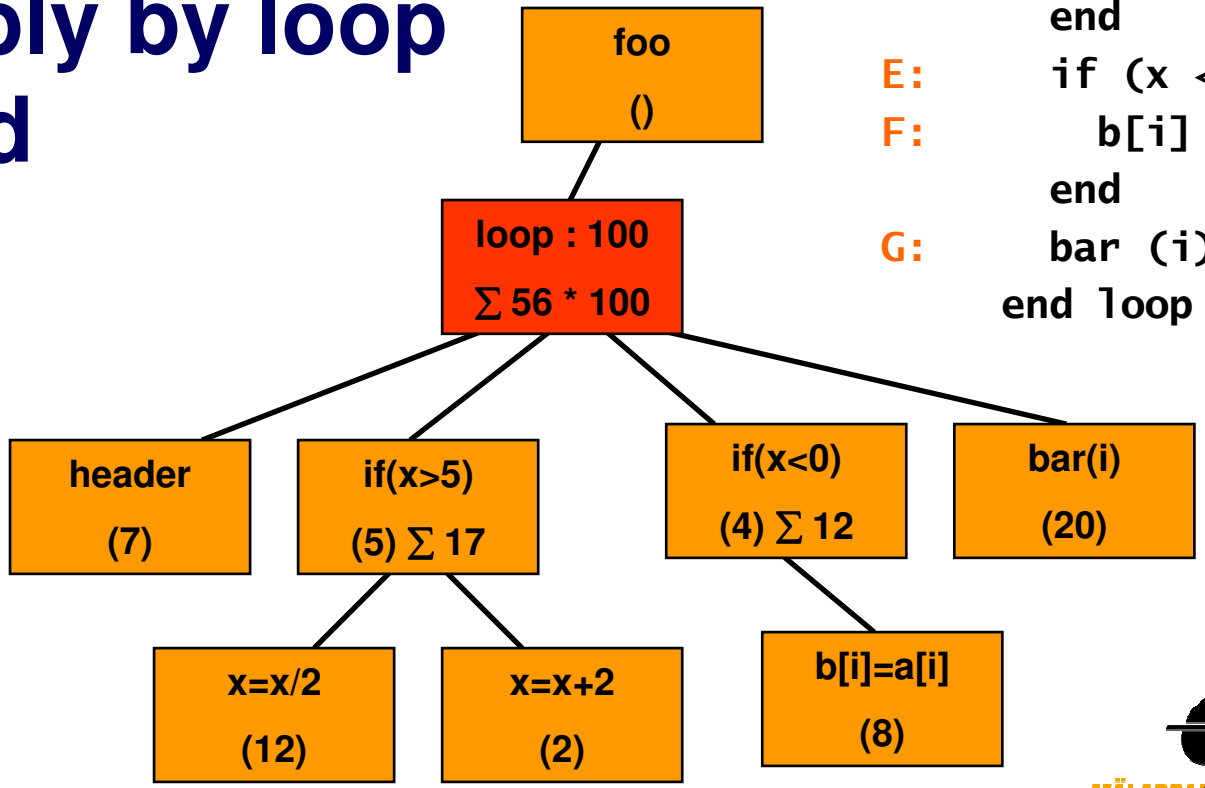
```
     foo(x):
A:    loop(i=1..100)
B:      if (x > 5) then
C:        x = x*2
        else
D:        x = x+2
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      bar (i)
      end loop
```



Tree-Based: LOOP

- Loop: sum the children
- Multiply by loop bound

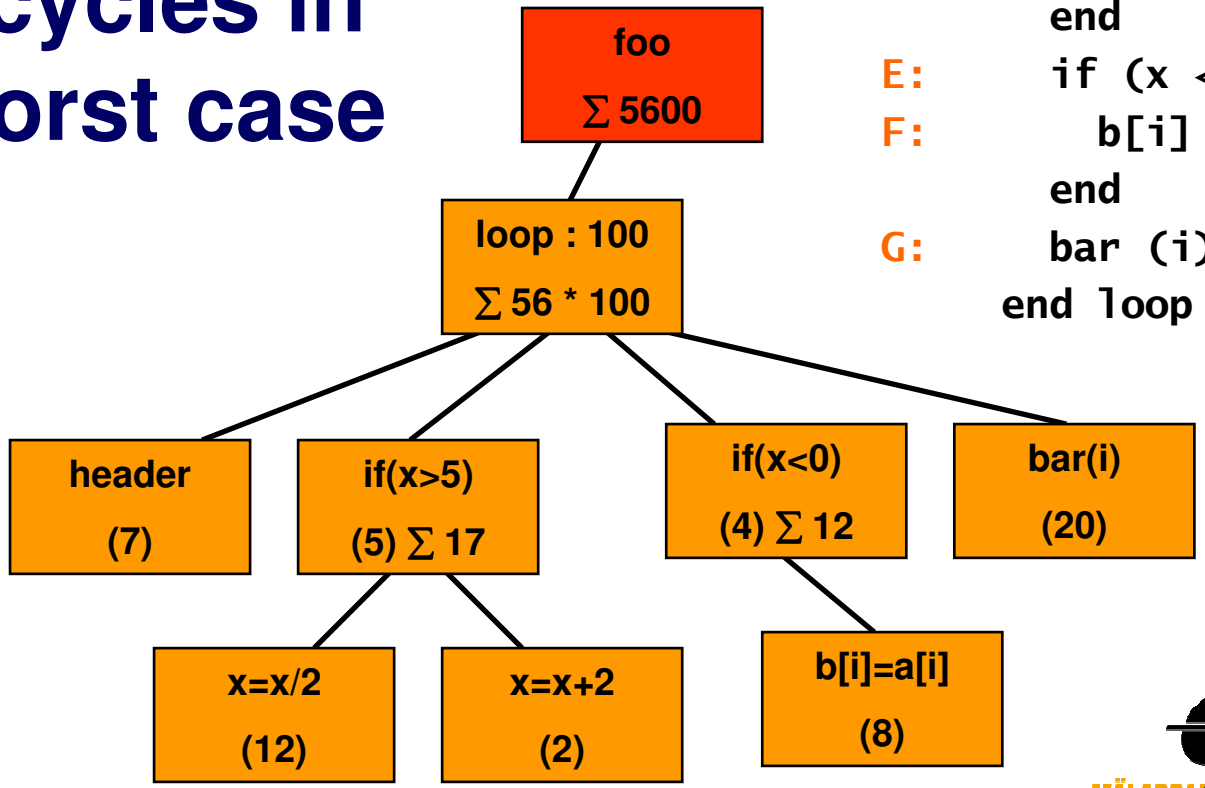
```
     foo(x):
A:    loop(i=1..100)
B:      if (x > 5) then
C:        x = x*2
        else
D:        x = x+2
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      bar (i)
      end loop
```



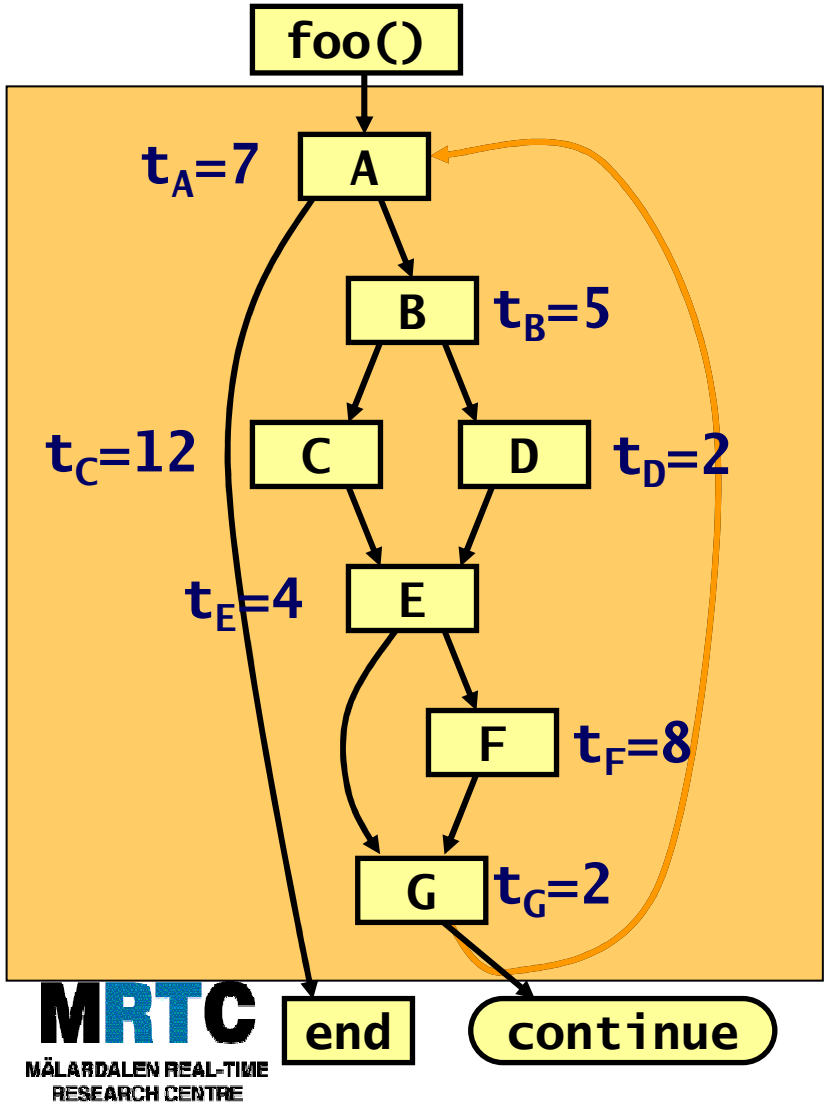
Tree-Based: Final result

- The function foo() will take 5600 cycles in the worst case

```
     foo(x):
A:    loop(i=1..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:      bar (i)
      end loop
```



Path-Based Calc



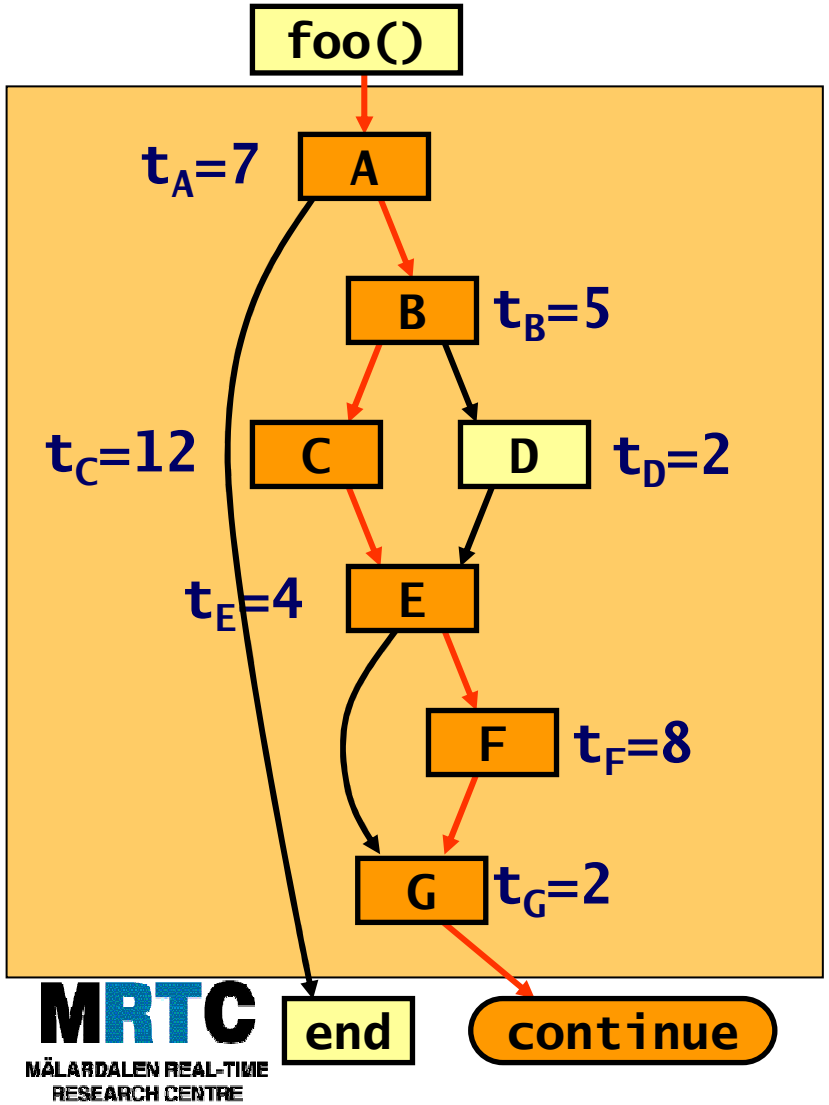

```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```

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

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Path-Based Calculation

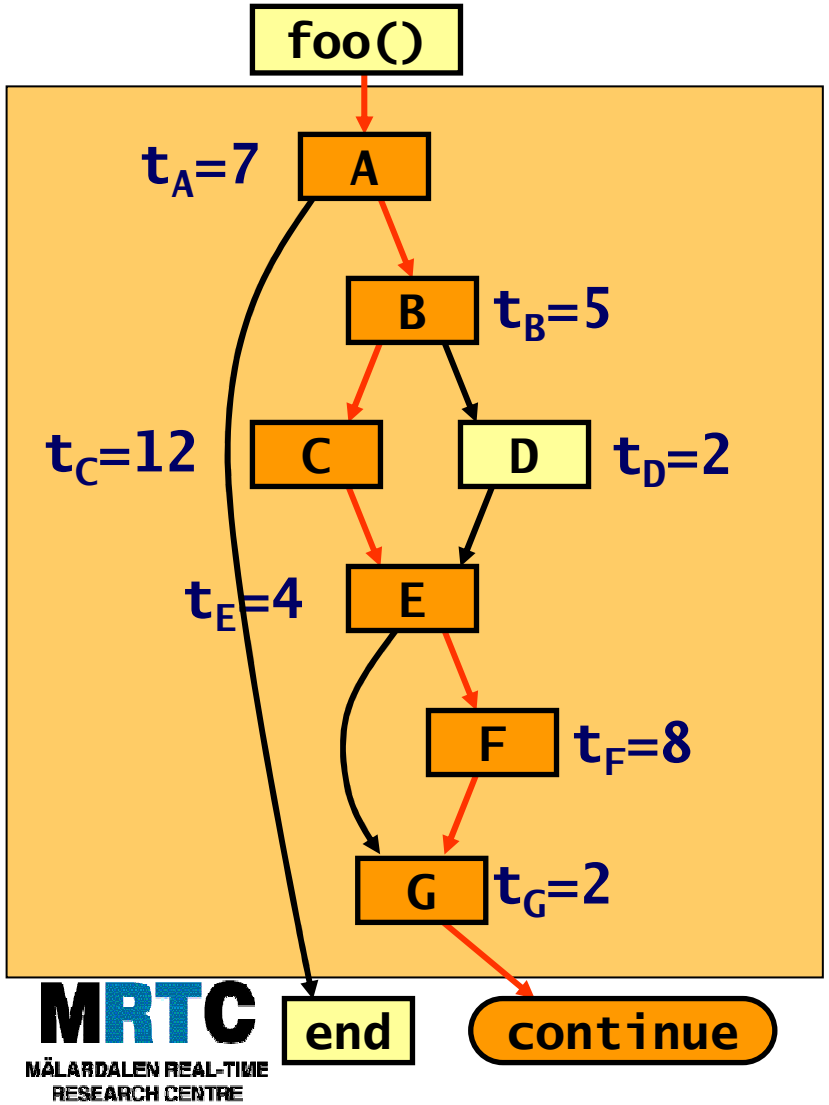


MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

- **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

MÄLARDALEN UNIVERSITY

Path-Based Calc



C and F can never execute together

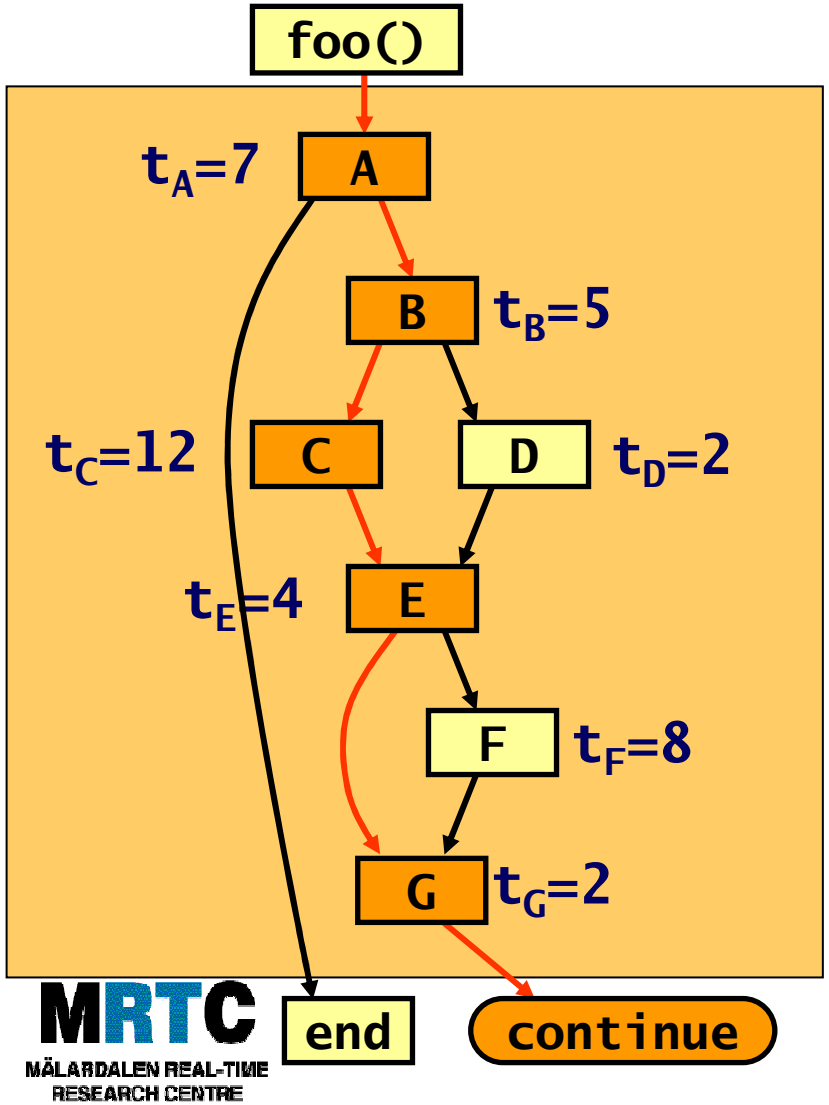
```
     foo(x,i):
A:    while(i < 100)
B:      if (x > 5) then
C:        x = x*2;
        else
D:        x = x+2;
        end
E:      if (x < 0) then
F:        b[i] = a[i];
        end
G:      i = i+1;
      end
```

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

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Path-Based Calc



C and F can never execute together

```
     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
```

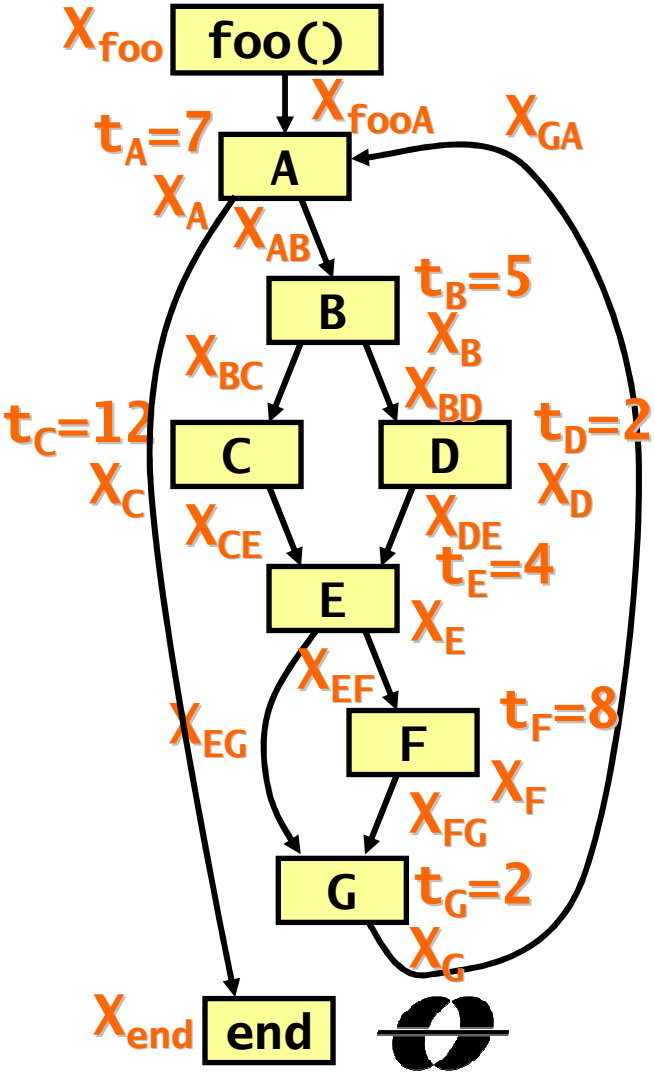
- **Infeasible path:**
  - A-B-C-E-F-G
  - Ignore, look for next
- **New longest path:**
  - A-B-C-E-G
  - 30 cycles
- **Total time:**
  - Total: 3000 cycles

MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Example: IPET Calculation

- IPET = Implicit path enumeration technique
  - Execution paths not explicitly represented
- Program model:
  - Nodes and edges
  - Timing info (t_{entity})
    - Node times: basic blocks
    - Edge times: overlap
  - Execution count (X_{entity})



MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

IPET Calculation

- **WCET=**

  $\max \sum(X_{entity} * t_{entity})$

  - Where each X_{entity} satisfies constraints

- **Constraints:**
  - Start & end condition
  - Program structure
  - Loop bounds
  - Other flow information

$X_{foo}=1$

$X_{end}=1$

$X_A=X_{fooA}+X_{GA}$

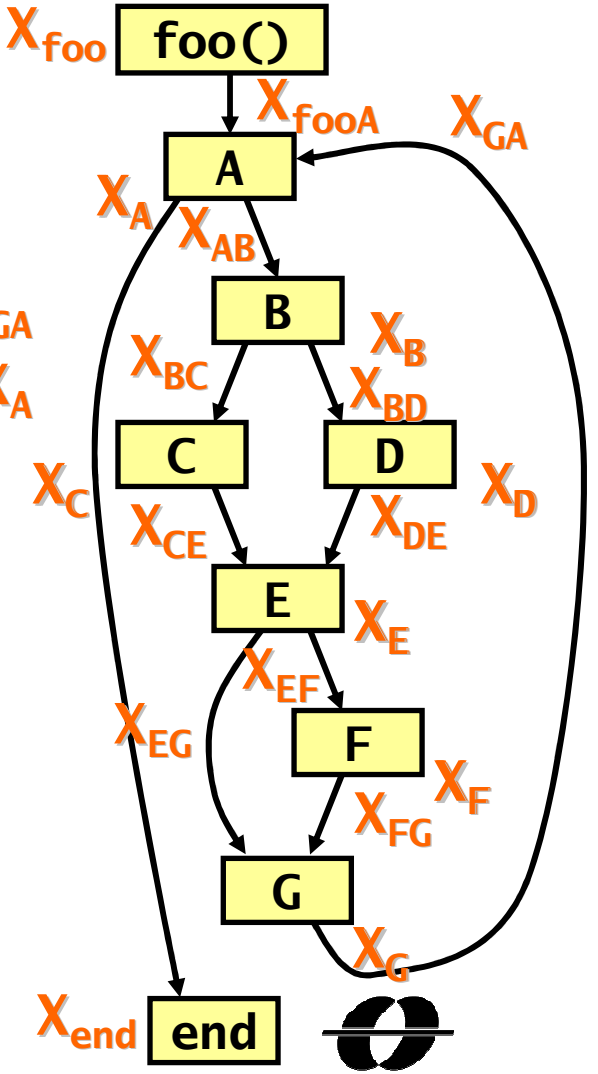
$X_{AB}+X_{Aend}=X_A$

$X_{BC}+X_{BD}=X_B$

$X_E=X_{CE}+X_{DE}$

$X_A<=100$

$X_C+X_F<=X_A$



MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

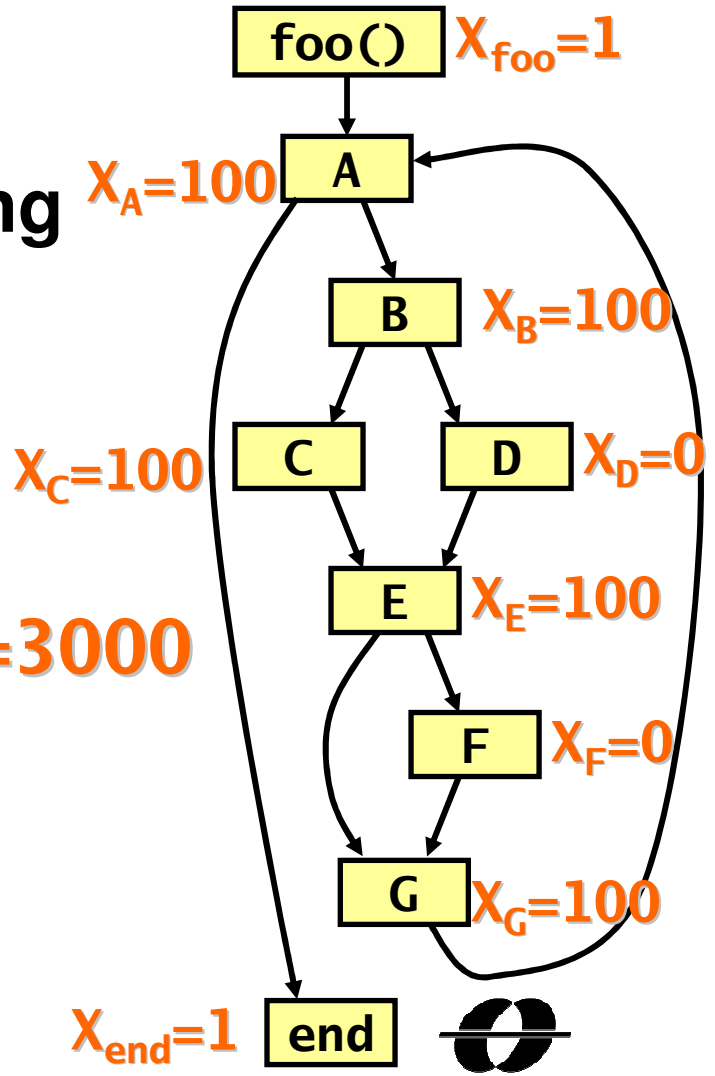
IPET Calculation

✶Solution methods:

- Integer linear programming
- Constraint satisfaction

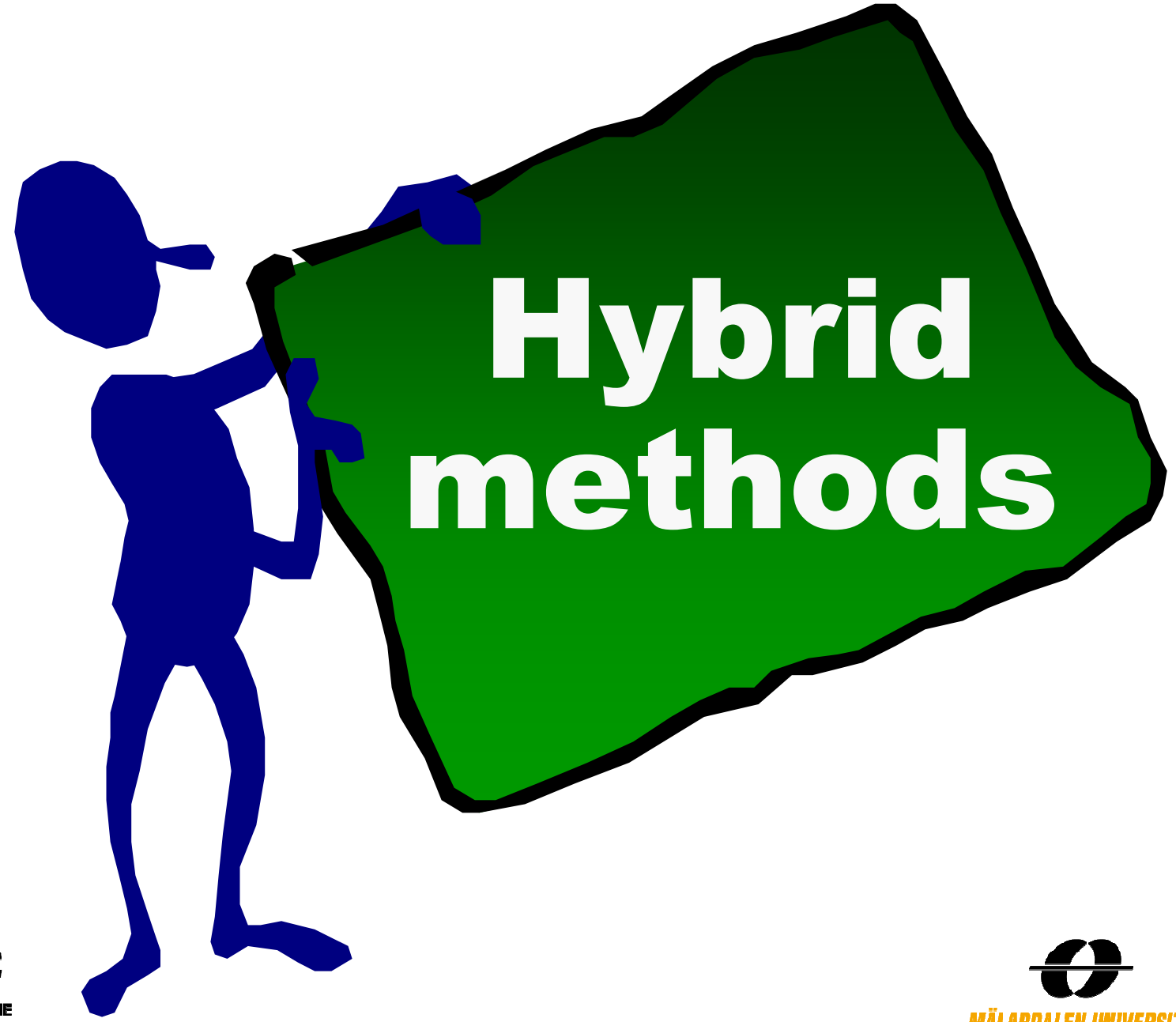
✶Solution:

- Counts for nodes and edges
- A WCET bound



MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY



MRTC
MÄLARDALEN REAL-TIME
RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Hybrid methods

- Combines measurement and static analysis
- Methodology:
  - Partition code into smaller parts
  - Identify & generate instrumentation points (ipoints) for code parts
  - Run program and generate ipoint traces
  - Derive time interval/distribution and flow info for code parts based on ipoint traces
  - Use code part's time interval/distribution and flow info to create a program WCET estimate
- Basis for RapiTime WCET analysis tool!

Example: loop bound derivation

```
int foo(int x) {
    write_to_port('A');
    int i = 0;
    while(i < x) {
        write_to_port('B');
        i++;
    }
}
```

Instrumentation code

Instrumentation code

Valid for an entry of foo()

- 3 example traces:
  - Run1: ABBBABBBBA
  - Run2: ABBAAABBA
  - Run3: ABBBBBBA
- Result (based on provided traces):
  - Lower loop bound: 0
  - Upper loop bound: 6

Example: function time derivation

```
int foo(int x) {
    write_to_port('A',TIME);
    int i = 0;
    while(i < x) {
        i++;
    }
    write_to_port('B',TIME);
}
```

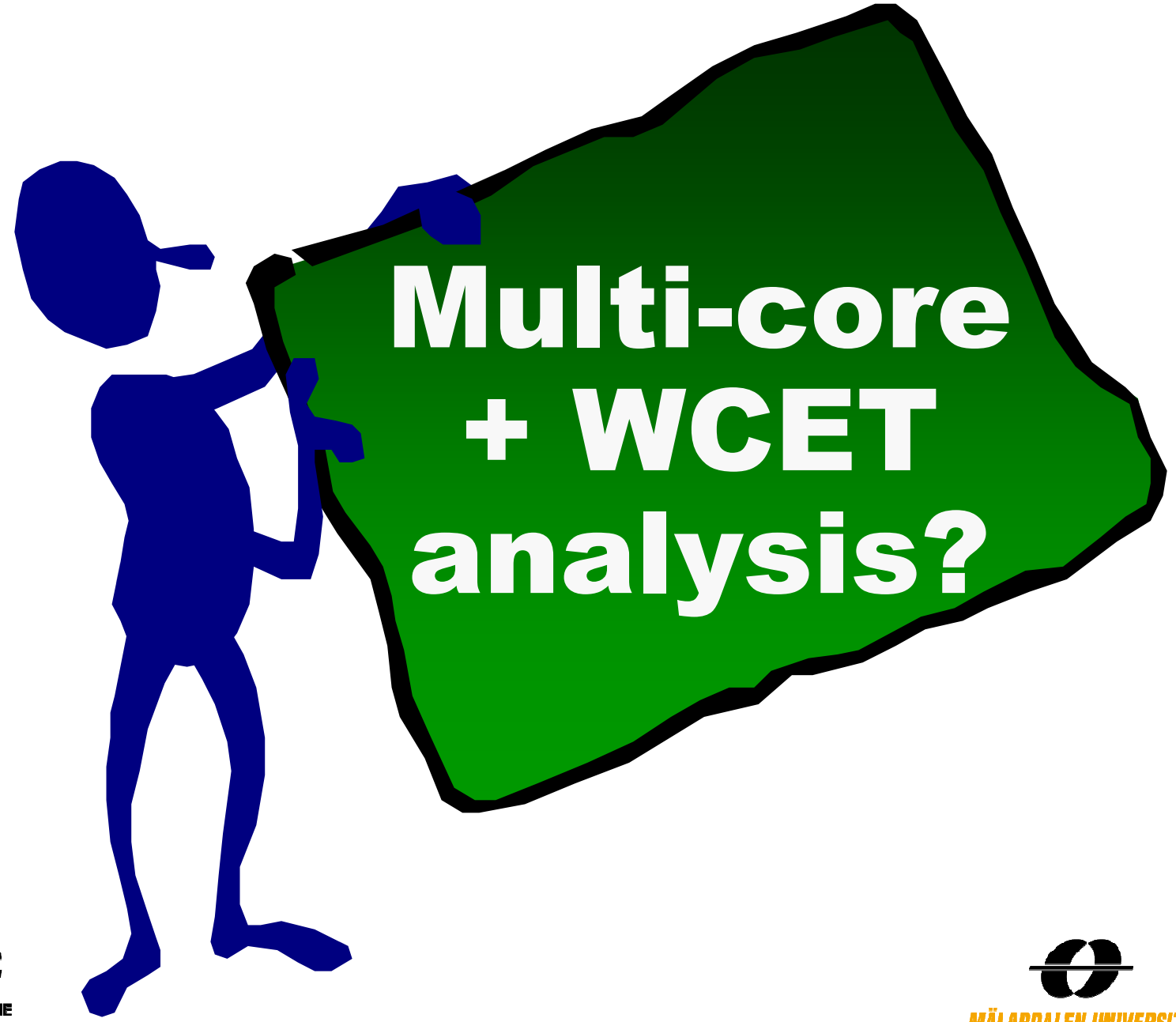
Instrumentation code extended with TIME macro

Realized as a short assembler snippet

- **Example trace:**
  - <A,72>,<B,156>, <A,2001>,<B,2191>, <A,2555>,<B,2661>
- **Result (based on provided trace):**
  - Min time foo: 84 (156-72=84)
  - Max time foo: 190 (2191-2001=190)

Notes: Hybrid methods

- + Testing and instrumentation already used in industry!
  - Known testing coverage criteria can be used
- + No hardware timing model needed!
  - Relatively easy to adapt analysis to new hardware targets
- – Is the resulting WCET estimate safe?
  - Have all costly software paths been executed?
  - Have all hardware effects been provoked/captured?
- – How much do instrumentation affect execution time?
  - Will timing behavior differ if they are removed?
  - Often constraints on where instrumentation points can be placed
  - Often limits on the amount of instrumentation points possible
  - Often limits on the bandwidth available for traces extraction
- – Are task switches/interrupts detected?
  - If not, derived timings may include them!

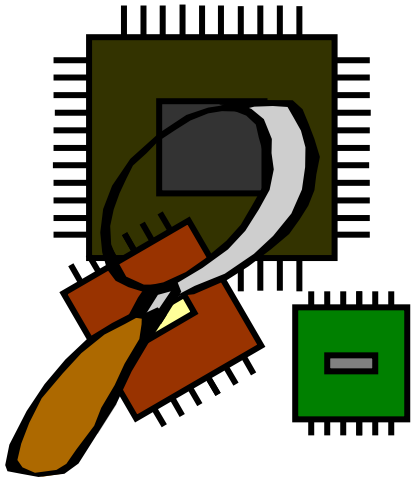


MRTC
MÄLARDALEN REAL-TIME
RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

Trends in Embedded HW

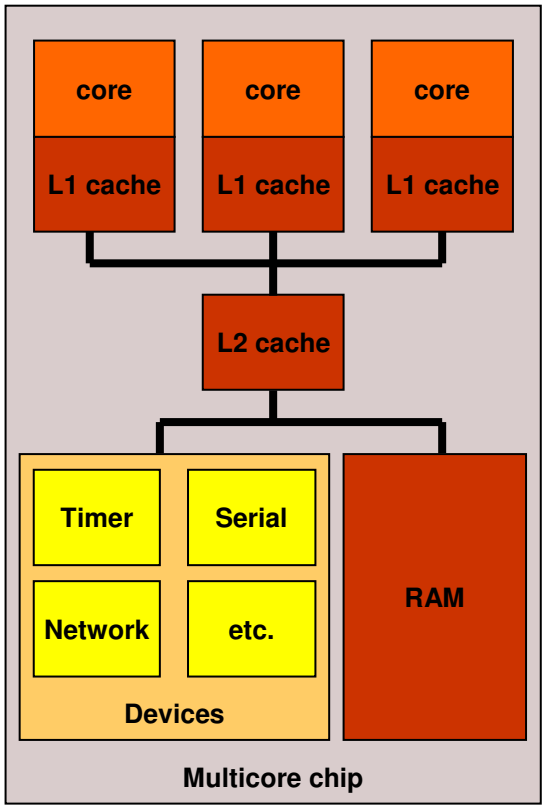
- **Trend: Large variety of ES HW platforms**
  - Not just one main processor type as for PCs
  - Many different HW configurations (memories, devices, …)
  - Challenge: How to make WCET analysis portable between platforms?
- **Trend: Increasingly complex HW features to boost performance**
  - Taken from the high-performance CPUs
  - Pipelines, caches, branch predictors, superscalar, out-of-order, …
  - Challenge: How to create safe and tight HW timing models?



- **Trend: Multi-core architectures**

Multi-core architectures

- Several (simple) CPUs on one chip
  - Increased performance & lower power
  - "SoC": System-on-a-Chip possible
- Explicit parallelism
  - Not hidden as in superscalar architectures
- Likely that CPUs will be less complex than current high-end processors
  - Good for WCET analysis!
- However, risk for more shared resources: buses, memories, …
  - Bad for WCET analysis!
  - Unrelated threads on other cores might use shared resources
- Multi-core might be ok if predictable sharing of common resources is somehow enforced

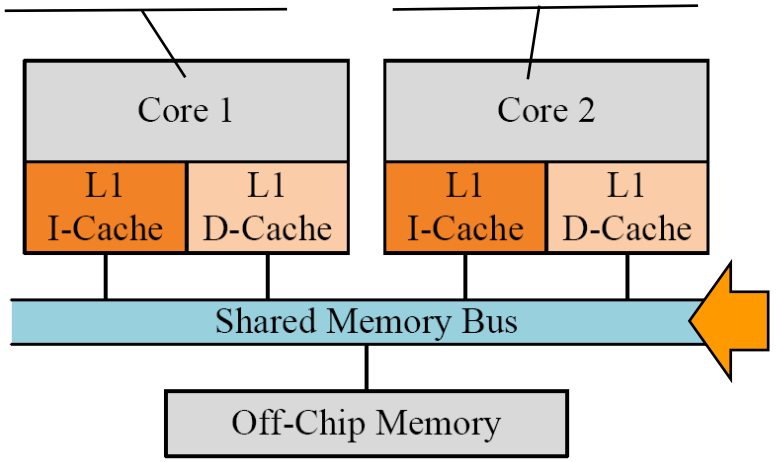


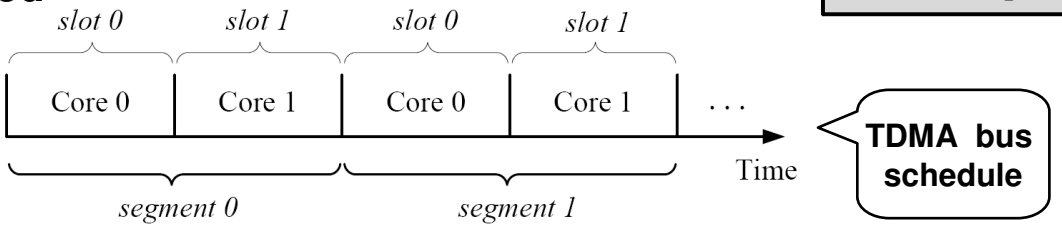
Example: shared bus

- **Example, dual core processor with private L1 caches and shared memory bus for all cores**
  - Each core runs its own code and task
- **Problem:**
  - Whenever t1 needs something from memory it may or may not collide with t2's accesses on the memory bus
  - Depends on <u>what</u> t1 and t2 accesses and <u>when</u> they accesses it
  - Large parallel state space to explore
- **Possible solution:**
  - Use deterministic (but potentially pessimistic) bus schedule, like TDMA
  - Worst-case memory bus delay can then be bounded

```
int t1_code {
  if(...) {
    ...
  }
  ...
}
```

```
int t2_code {
  ...
  while(...) {
    ...
  }
}
```



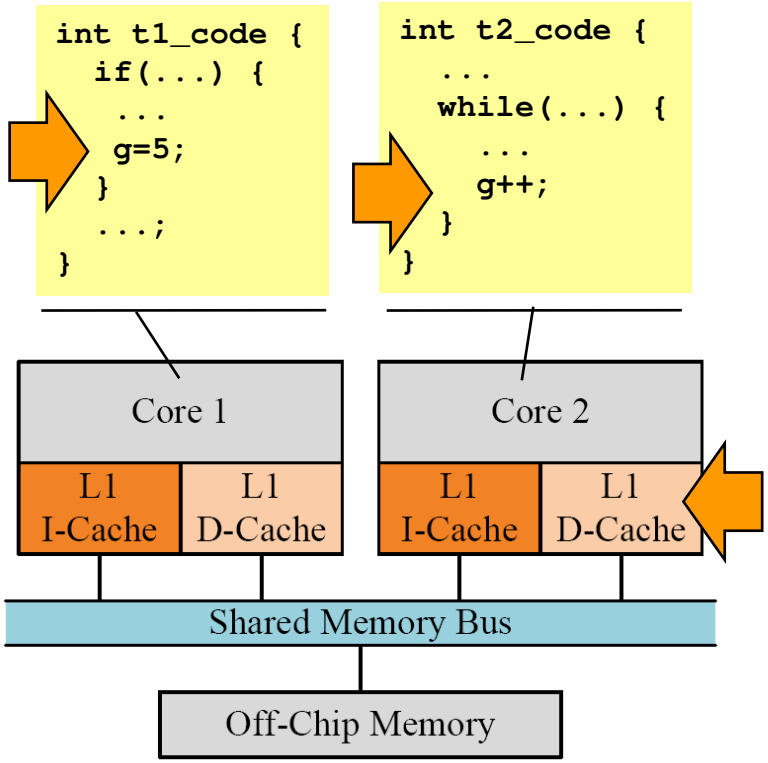


MRTC
MÄLARDALEN REAL-TIME RESEARCH CENTRE

MÄLARDALEN UNIVERSITY

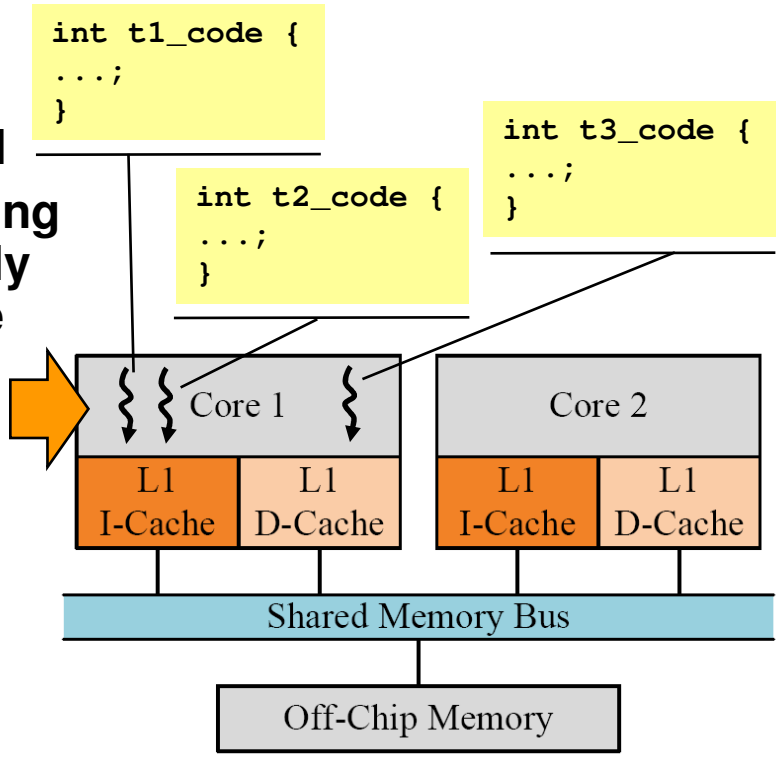
Example: shared memory

- ES often programmed using shared memory model
  - t1 and t2 may communicate/synchronize using shared variables
- Problem:
  - When t1 writes g, memory block of g is loaded into core1's d-cache
  - Similarly, when t2's writes g, memory block of g moved to t2's d-cache (and t1's block is invalidated)
- May give a large overhead
  - Much time can be spent moving memory blocks in between caches (ping-pong)
  - Hidden from programmer - HW makes sure that cache/memory content is ok
  - False sharing – when tasks accesses different variables, but variables are located in same memory block
- Possible solutions:
  - Constrain task's accesses to shared memory (e.g. single-shot task model)



Example: multithreading

- Common on high-order multi-cores and GPUs
- Core run multiple threads of execution in parallel
  - Parts of core that store state of threads (registers, PC, ..) replicated
  - Core's execution units and caches shared between threads
- Benefits
  - Hides latency – when one thread stalls another may execute instead
  - Better utilization of core's computing resources – one thread usually only use a few of them at the same time
- Problems
  - Hard to get timing predictability
  - Instructions executing and cache content depends dynamically on state of threads, scheduler, etc.



Trends in Embedded SW

- Traditionally: embedded SW written in C and assembler, close to hardware
- Trend: size of embedded SW increases
  - SW now clearly dominates ES development cost
  - Hardware used to dominate
- Trend: more ES development by high-level programming languages and tools
  - Object-oriented programming languages
  - Model-based tools
  - Component-based tools

Increase in embedded SW size

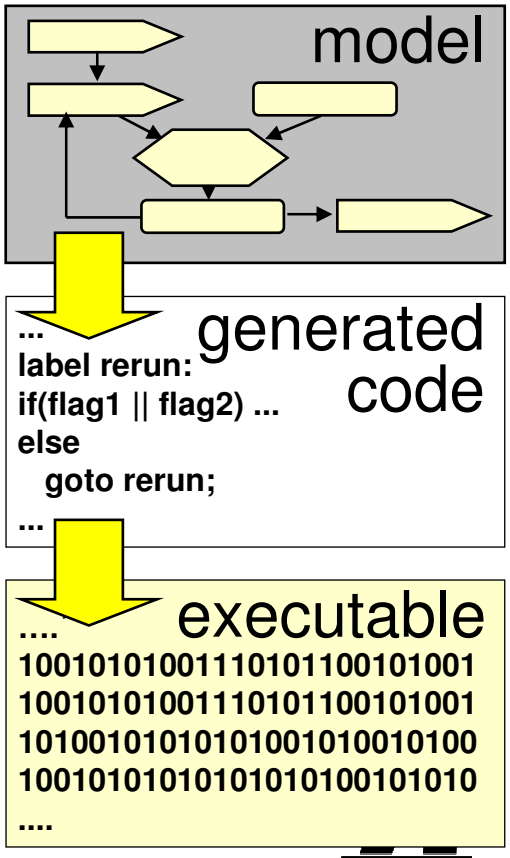
- More and more functionality required
  - Most easily realized in software
- Software gets more and more complex
  - Harder to identify the timing critical part of the code
  - Source code not always available for all parts of the system, e.g. for SW developed by subcontractors
- Challenges for WCET analysis:
  - Scaling of WCET analysis methods to larger code sizes
    - Better visualization of results (where is the time spent?)
  - Better adaptation to the SW development process
    - Today's WCET analysis works on the final executable
    - Challenge: how to provide reasonable precise WCET estimates at early development stages

Higher-level prog. languages

- Typically object-oriented: C++, Java, C#, …
- Challenges for WCET analysis:
  - Higher use of dynamic data structures
    - In traditional ES programming all data is statically allocated during compile time
  - Dynamic code, e.g., calls to virtual methods
    - Hard to analyze statically (actual method called may not be known until run-time)
  - Dynamic middleware:
    - Run-time system with GC
    - Virtual machines with JIT compilation

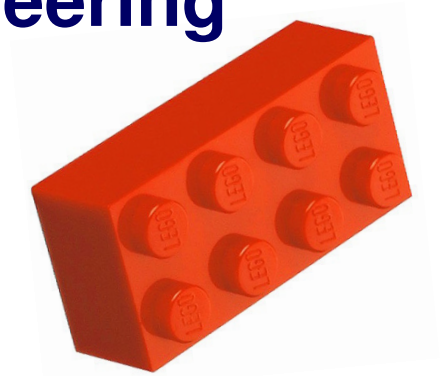
Model-based design

- More embedded system code generated by higher-level modeling and design tools
  - RT-UML, Ascet, Targetlink, Scade, ...
- The resulting code structure depends on the code generator
  - Often simpler than handwritten code
- Possible to integrate such tools with WCET analysis tools
  - The analysis can be automated
  - E.g., loop bounds can be provided directly by the modeling tool
- Hard to provide reliable timing on modeling level



Component-based design

- Very trendy within software engineering
- General idea:
  - Package software into reusable *components*
  - Build systems out of prefabricated components, which are “glued together”
- WCET analysis challenges:
  - How to reuse WCET analysis results when some settings have changed?
  - How to analyze SW components when not all information is available?
  - Are WCET analysis results composable?



The End!

For more information:

www.mrtc.mdh.se/projects/wcet