# TDDI08: Embedded Systems Design

# TDTS07: System Design and Methodology (Lesson 1)

# Contacts

- Soheil Samii (lectures)
  - Office: Building B 329:220
  - Email: soheil.samii@liu.se
- Yungang Pan (lessons & labs)
  - Office: Building B 329:202
  - Email: yungang.pan@liu.se

# Outline

- Today
  - Organization
  - Lab 1 (TDTS07 & TDDI08):
    - Modeling and Simulation with SystemC
  - Lab 2 (TDDI08) / Lab 3 (TDTS07):
    - Design-space Exploration with MPARM
- Next time
  - Lab 2 (TDTS07)
    - Formal verification with UPPAAL

# TDDI08 - Organization

- Lab groups
  - Webreg groups A and B
- Web page
  - https://www.ida.liu.se/~TDDI08
  - Check for detailed information and links to tutorials
- Organization
  - 1 lesson (this one)
  - 7 two-hour lab sessions
- Lab assignments
  1. Modeling and simulation with SystemC (4-5 sessions)
  2. Design-space exploration with MPARM (2-3 sessions)

# TDTS07 - Organization

- Lab groups
  - Webreg groups A and B
- Web page
  - https://www.ida.liu.se/~TDTS07
  - Check for detailed information and links to tutorials
- Organization
  - 2 lessons (including this one)
  - 10 two-hour lab sessions
- Lab assignments
  1. Modeling and simulation with SystemC (3-4 sessions)
  2. Formal verification with UPPAAL (4-5 sessions)
  3. Design-space exploration with MPARM (2 sessions)

# Organization

- Choose a lab partner and sign up in Webreg
  - https://www.ida.liu.se/webreg3/TDDI08-2024-1/LAB
  - https://www.ida.liu.se/webreg3/TDTS07-2024-1/LAB
  - Deadline for the registration: **January 24**
  - Register as soon as possible
- Deadline for the labs: **March 6**
  - This is the last day for handing in (emailing) lab reports
  - After the deadline, your teaching assistant will correct the remaining lab reports at his convenience
- Lab rules
  - https://www.ida.liu.se/labs/eslab/lab_rules
  - Read them!

# Structure

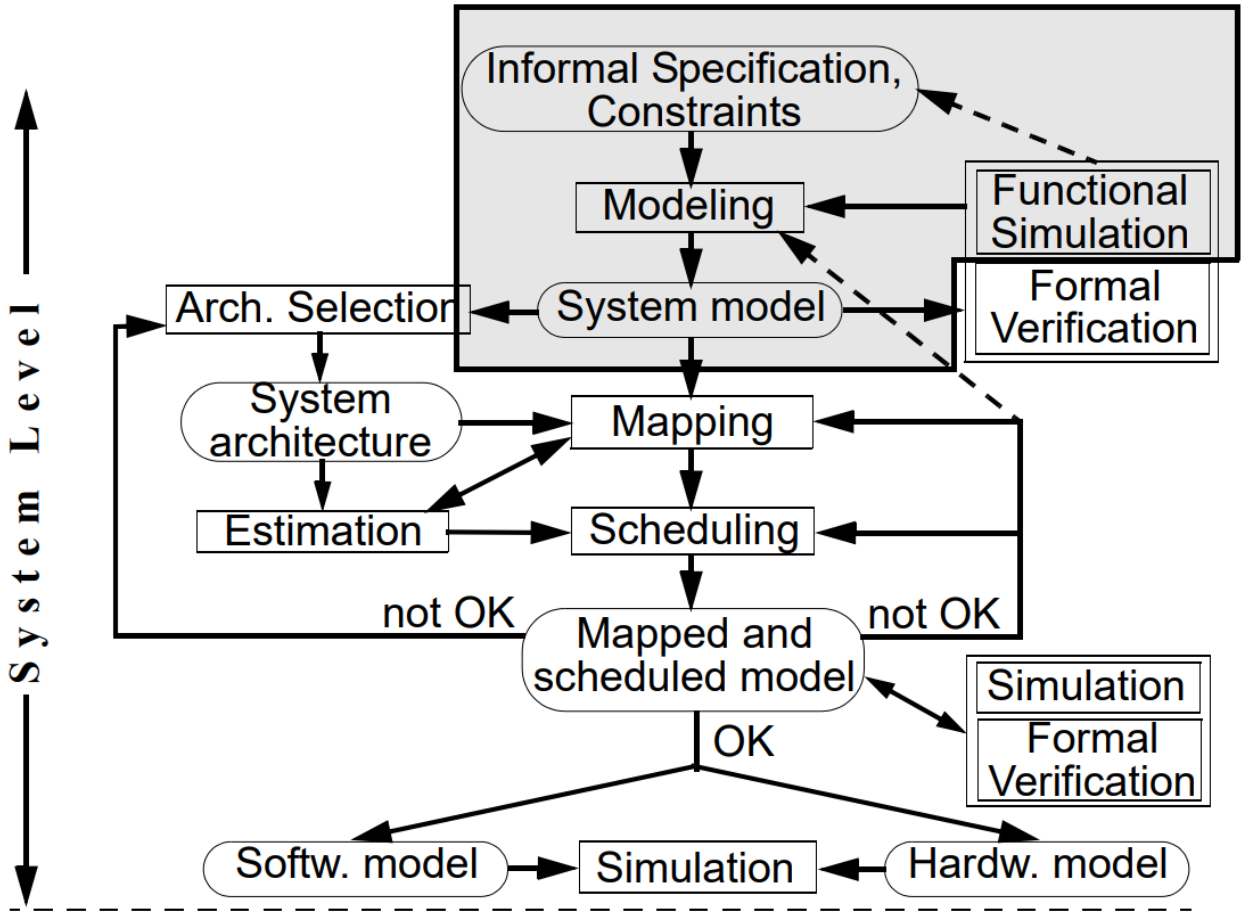
1. Modeling and simulation with SystemC (TDDI08 & TDTS07)
2. Formal verification with UPPAAL (TDTS07)
3. Design-space exploration with MPARM (TDDI08 & TDTS07)

- Each lab has a tutorial. Read it and be prepared before you attend the lab session.
- https://www.ida.liu.se/~TDDI08/labs/index.en.shtml
- https://www.ida.liu.se/~TDTS07/labs/index.en.shtml

# Introduction to Lab 1

# Modeling and Simulation with SystemC

- **Modeling and simulation with System C**



# Simulation

- Based on an executable model of the system
- Generate inputs and observe outputs
- Permits a quick but shallow evaluation
- Good for detecting crude errors
- Not good for finding subtle bugs

# SystemC

- Comparable to VHDL and Verilog
  - Unified hardware-software design language
  - Contains structures for modeling hardware components and their interaction
  - Comes with a simulation kernel
- What do we need to model systems?
  - time
  - modules
  - concurrent processes
  - events
  - channels
  - ports

# SystemC: Time

- Data type **`sc_time`** (a C++ class)
- Use like an ordinary basic data type (**`int`**, **`double`**)
  - `sc_time t1(9, SC_MS);`
  - `sc_time t2 = sc_time(5, SC_SEC);`
  - `if (t1<t2) cout << t1*3 << endl << t2+t2;`
  - Many of standard operators are defined for **`sc_time`**
- Based on 64-bits unsigned integer values
  - The representable time is limited (discrete time)
  - Depends on the time resolution
    - Default: 1 picosecond
    - Can be set by the user through the function **`sc_set_time_resolution`**

# SystemC: Modules

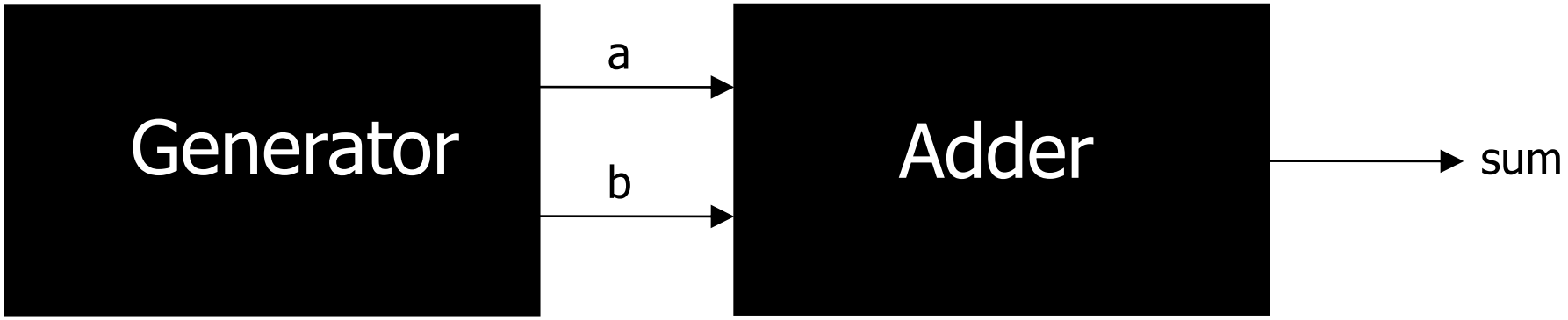
- Basic building blocks in SystemC
  - Contains ports, concurrent processes, internal data structures, channels, etc.
- Created with the macro **`SC_MODULE`**
- Concurrent processes (**`SC_THREAD`** or **`SC_METHOD`**)
  - Use **`wait`** statements to advance time (or event notification)
  - Sensitive to events (**`sc_event`**) or value changes in channels
- Input and output ports to communicate with the environment

# Example: Adder

# Adder Module

```
#include <systemc.h>
#include <iostream>

using std::cout;
using std::endl;

SC_MODULE(Adder) {
  sc_in<int> a_p;
  sc_in<int> b_p;
  sc_out<int> sum_p;
  sc_event print_ev;

  void add_method() {
    sum_p = a_p + b_p;
    print_ev.notify(SC_ZERO_TIME);
  }
  …
  …
  void print_method() {
    cout << sc_time_stamp()
         << ":Sum=" <<sum_p
         << endl;
  }

  SC_CTOR(Adder) {
    sum_p.initialize(0);
    SC_METHOD(add_method);
    sensitive << a_p << b_p;
    SC_METHOD(print_method);
    dont_initialize();
    sensitive << print_ev;
  }
};
```

# Generator



# Generator Module

```
SC_MODULE(Generator) {
  sc_out<int> a_p;
  sc_out<int> b_p;

  void gen_thread() {
    for (;;) {
      wait(1, SC_SEC);
      a_p = a_p + 1;
      b_p->write(b_p->read() + 1);
    }
  }

  SC_CTOR(Generator) {
    a_p.initialize(0);
    b_p.initialize(0);
    SC_THREAD(gen_thread);
  }
};
```

# Test Bench

```
// Definition of an input generator
int sc_main(int argc, char *argv[]) {
  sc_set_default_time_unit(1, SC_SEC);
  sc_signal<int> a_sig, b_sig, sum_sig; // create channels
  Adder adder_module("Adder_1");        // create an instance
  adder_module(a_sig, b_sig, sum_sig);  // connect ports to
                                        // channels
  Generator gen("Generator_1");
  gen(a_sig, b_sig);
  sc_start(30, SC_SEC);
  return 0;
}
```

# Simulation Run

```
$ ./adder.x

            SystemC 2.1.v1 --- Dec 22 2014 16:12:32
        Copyright (c) 1996-2005 by all Contributors
                    ALL RIGHTS RESERVED
0 s: Sum=0
1 s: Sum=2
2 s: Sum=4
3 s: Sum=6
4 s: Sum=8
5 s: Sum=10
6 s: Sum=12
7 s: Sum=14
8 s: Sum=16
9 s: Sum=18
10 s: Sum=20
11 s: Sum=22
…
```

# SystemC: Simulator Kernel

1. **Initialize**: each process is executed once; it's possible to disable this phase for methods.

```
#include <systemc.h>
#include <iostream>

using std::cout;
using std::endl;

SC_MODULE(Adder) {
  sc_in<int> a_p;
  sc_in<int> b_p;
  sc_out<int> sum_p;
  sc_event print_ev;

  void add_method() {
    sum_p = a_p + b_p;
    print_ev.notify(SC_ZERO_TIME);
  }
  …
  void print_method() {
    cout << sc_time_stamp()
         << ":Sum=" <<sum_p
         << endl;
  }

  SC_CTOR(Adder) {
    sum_p.initialize(0);
    SC_METHOD(add_method);
    sensitive << a_p << b_p;
    SC_METHOD(print_method);
    dont_initialize();
    sensitive << print_ev;
  }
};
```

# SystemC: Simulator Kernel

1. **Initialize**: each process is executed once; it's possible to disable this phase for methods.
2. **Evaluate**: select a ready-to-run process and execute or resume it; immediate notification may happen (e.notify()).
3. Repeat Step 2 until there are no more processes to run.

```
#include <systemc.h>
#include <iostream>

using std::cout;
using std::endl;

SC_MODULE(Adder) {
  sc_in<int> a_p;
  sc_in<int> b_p;
  sc_out<int> sum_p;
  sc_event print_ev;

  void add_method() {
    sum_p = a_p + b_p;
    print_ev.notify(SC_ZERO_TIME);
  }
  …
  void print_method() {
    cout << sc_time_stamp()
         << ":Sum=" <<sum_p
         << endl;
  }

  SC_CTOR(Adder) {
    sum_p.initialize(0);
    SC_METHOD(add_method);
    sensitive << a_p << b_p;
    SC_METHOD(print_method);
    dont_initialize();
    sensitive << print_ev;
  }
};
```

```
0 s: Sum=0
```

# SystemC: Simulator Kernel

1. **Initialize**: each process is executed once; it's possible to disable this phase for methods.
2. **Evaluate**: select a ready-to-run process and execute or resume it; immediate notification may happen (e.notify()).
3. Repeat Step 2 until there are no more processes to run.
4. **Update**: values assigned to channels in the previous evaluation cycle are updated.

# SystemC: Simulator Kernel

1. **Initialize**: each process is executed once; it's possible to disable this phase for methods.
2. **Evaluate**: select a ready-to-run process and execute or resume it; immediate notification may happen (e.notify()).
3. Repeat Step 2 until there are no more processes to run.
4. **Update**: values assigned to channels in the previous evaluation cycle are updated.
5. Steps 2–4 are a *delta-cycle*; if Step 2 or 3 resulted in delta event notifications (e.notify(0) or wait(0)), go to Step 2 *without* advancing the simulation time.

```
void add_method() {
  sum_p = a_p + b_p;
  print_ev.notify(SC_ZERO_TIME);
}
```

# SystemC: Simulator Kernel

1. **Initialize**: each process is executed once; it's possible to disable this phase for methods.
2. **Evaluate**: select a ready-to-run process and execute or resume it; immediate notification may happen (e.notify()).
3. Repeat Step 2 until there are no more processes to run.
4. **Update**: values assigned to channels in the previous evaluation cycle are updated.
5. Steps 2–4 are a *delta-cycle*; if Step 2 or 3 resulted in delta event notifications (e.notify(0) or wait(0)), go to Step 2 *without* advancing the simulation time.
6. Advance to the next time with pending events.
7. Determine processes ready to run and go to Step 2.

```
void gen_thread() {
  for (;;) {
    wait(1, SC_SEC);
    a_p = a_p + 1;
    b_p->write(b_p->read() + 1);
  }
}
```

- Concurrent processes (**`SC_THREAD`** or **`SC_METHOD`**)
  - Use **`wait`** statements to advance time (or event notification)

# SystemC: Delta-cycle

```
// Inside a process
sc_signal<int> sig_int;
// Assume current value of sig_int is 0
sig_int.write(1);
int value = sig_int.read();
cout << value << endl;  ----------->  0
wait(SC_ZERO_TIME);
value = sig_int.read();
cout << value << endl;  ----------->  1
```

# Run the Example

- Copy the example to your home directory
  - /courses/TDTS07/tutorials/systemc/adder
  - adder.cc (implements the system)
  - Makefile (helps you compile and build the program)
- Type **`make`** in the command line
  - Creates an executable **`adder.x`**
- Type **`./adder.x`** to run the executable
- Study the source code together with the tutorial

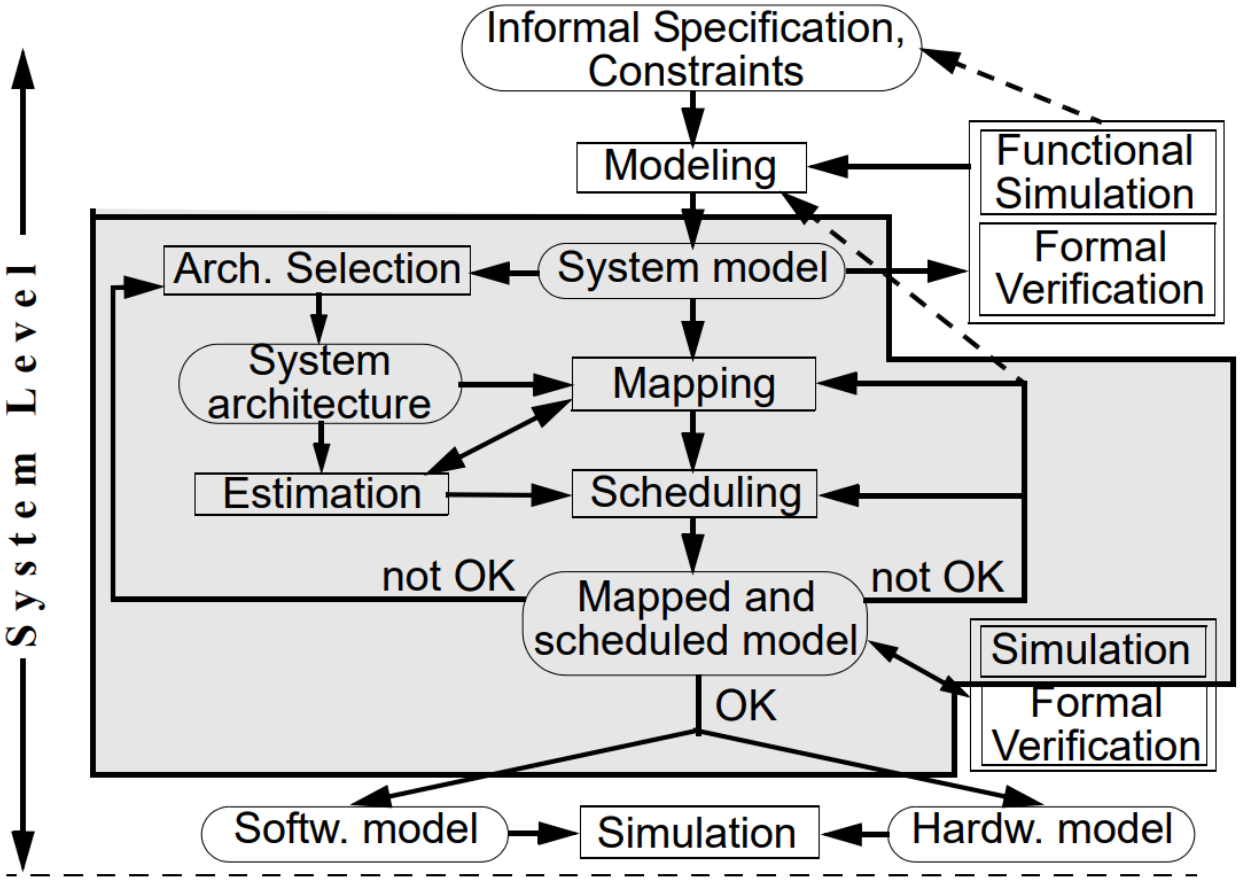
# Lab Assignment

- Study the lab material linked from the web pages
- There you will find the lab assignment
  - Design and implement a traffic light controller
- For further details
  - SystemC Language Reference Manual
  - http://accellera.org

# Introduction to Lab 2 / Lab 3
# Design-space Exploration with MPARM

# Lab 2

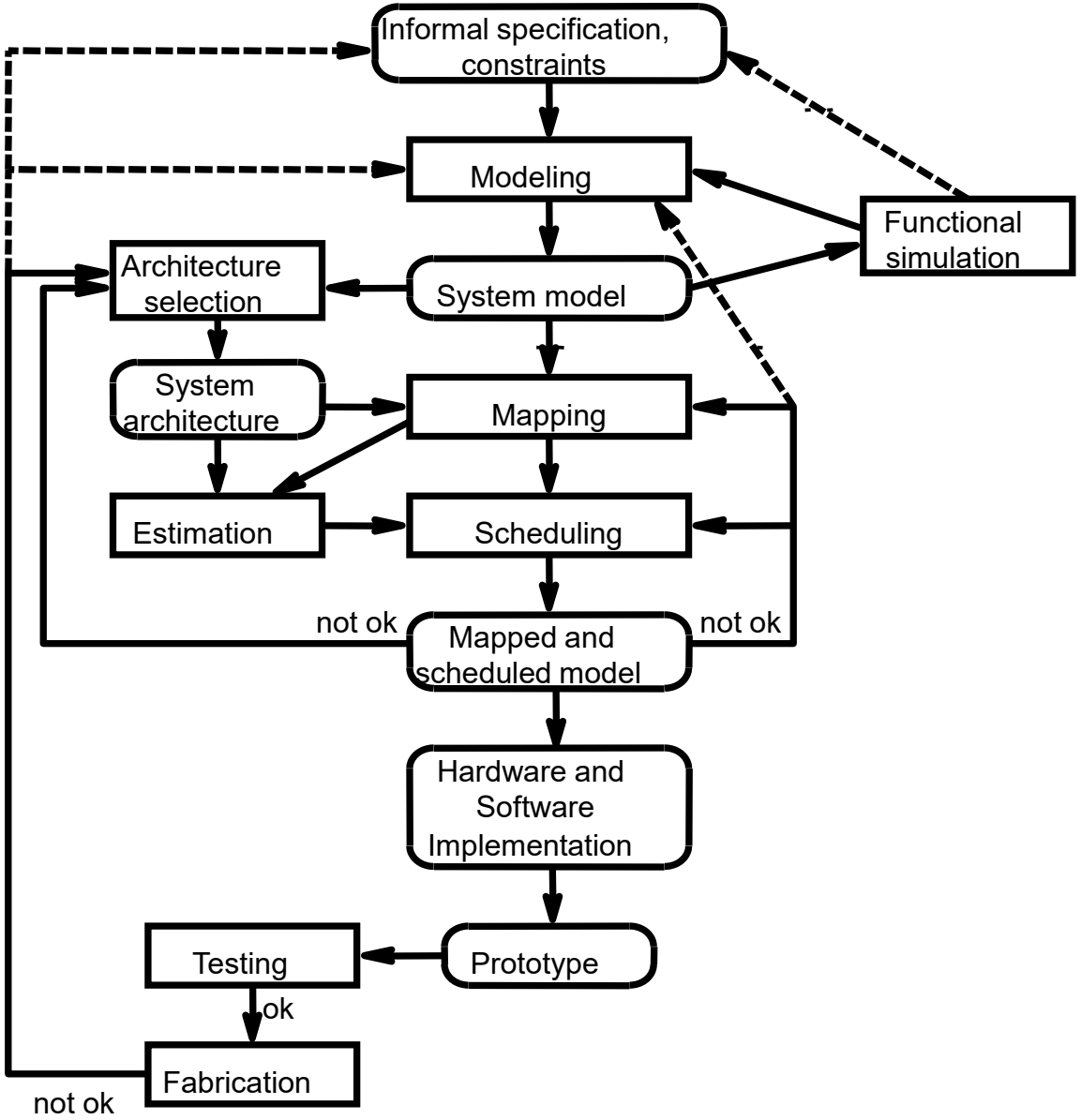
- **Design space exploration with an MPARM simulator.**



# Outline

- System-design flow
- Hardware and software
- Design-space exploration

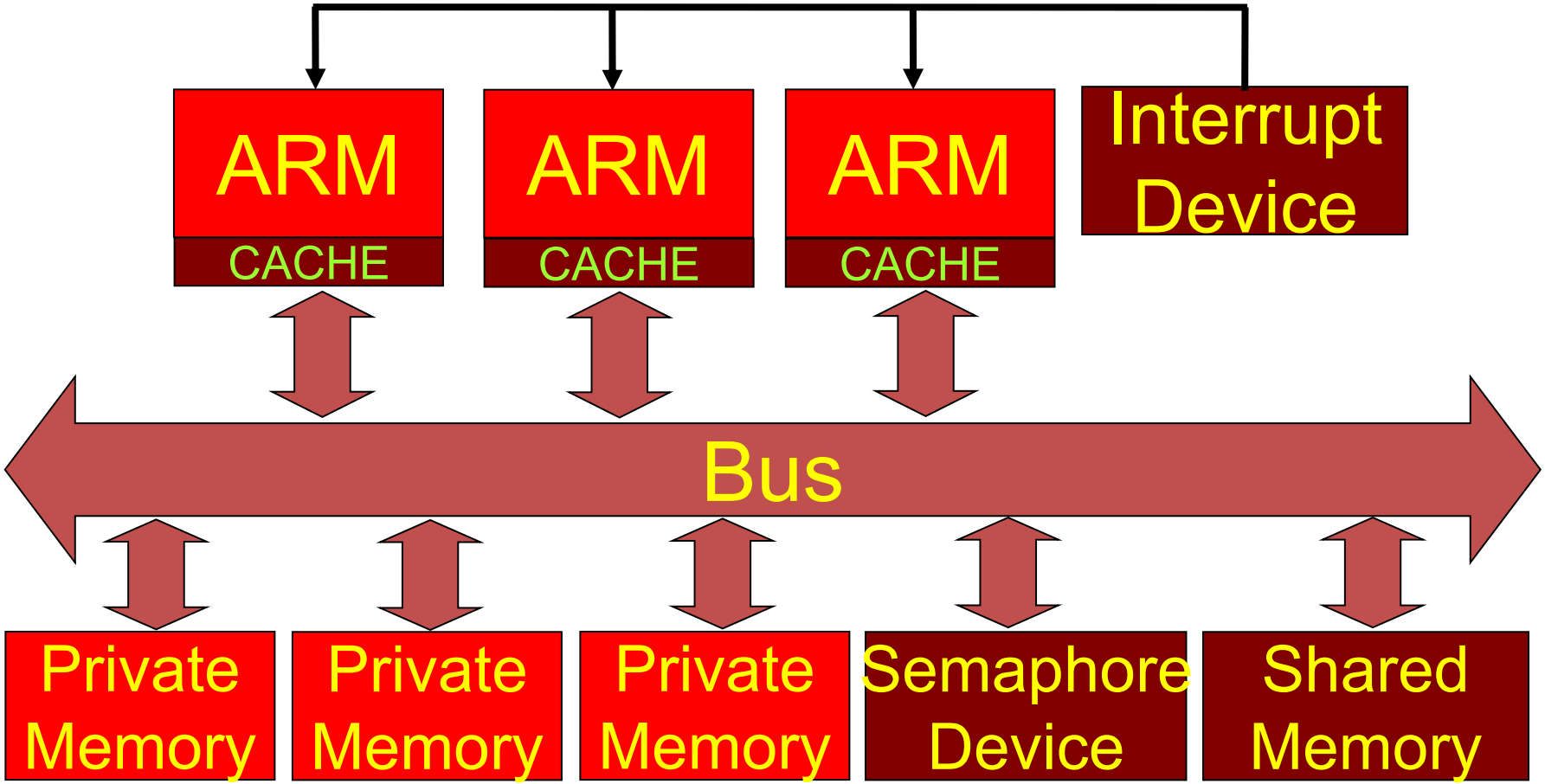
# System-design Flow



# MPARM: Motivation

- Cycle-accurate simulation of the system at hand
- Provides various statistics such as the number of clock cycles, bus utilization, cache efficiency, and energy/power consumption of the components
- Helps to obtain a correct and efficient product

# MPSoC Architecture



# MPARM: Hardware

- ARM7 processors (up to eight)
- Variable frequency (dynamic and static)
- Instruction and data caches
- Scratchpad memory
- Private memory
- Shared memory
- Communication bus
- Read more in:
  - /courses/**TDTS07**/sw/mparm/MPARM/doc
  - simulator_statistics.txt

# MPARM: Software

- Cross-compiler toolchain for building software
- No operating system
- Small set of functions (such as WAIT and SIGNAL)

# MPARM: Usage

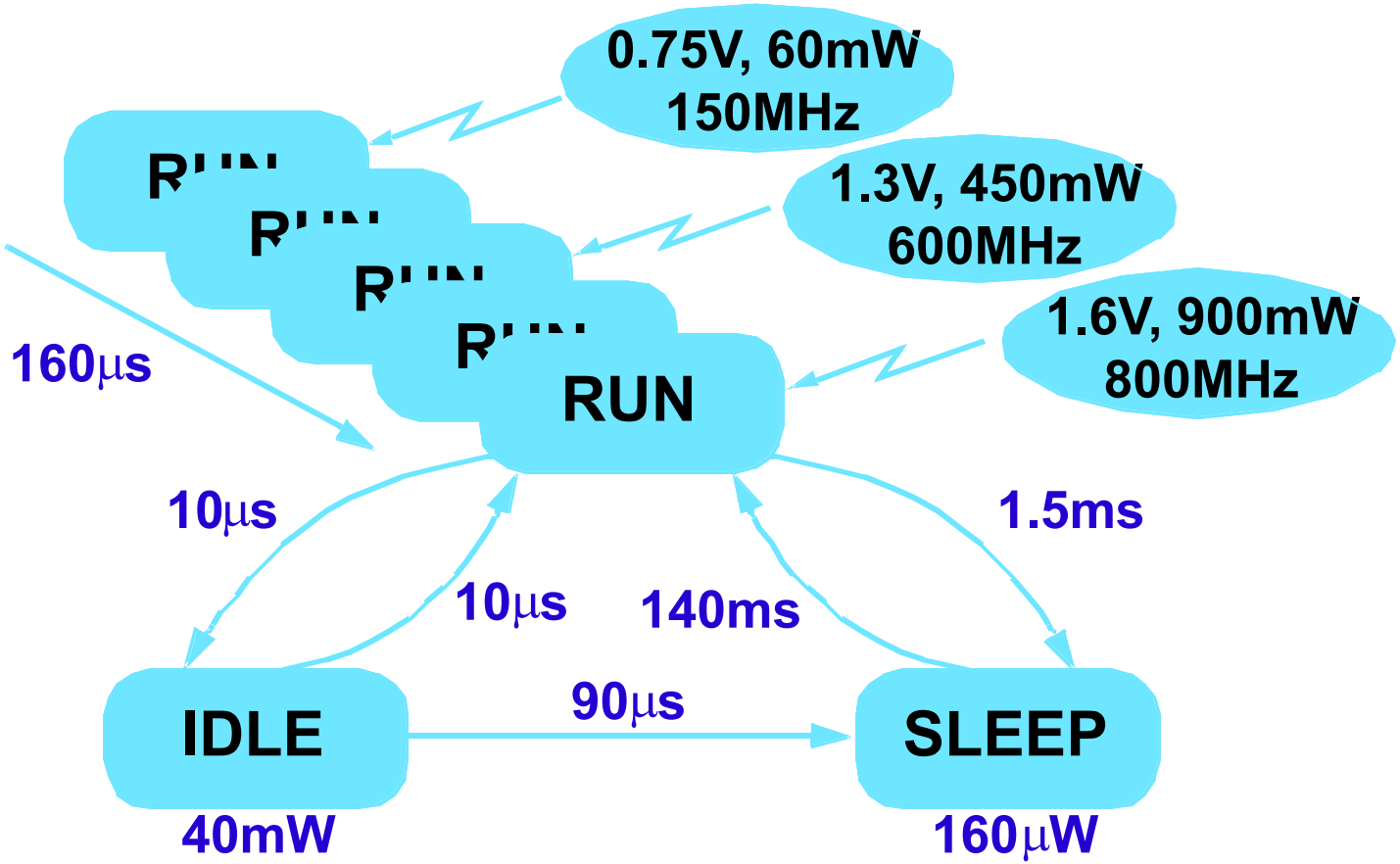
- mpsim.x -c2 — run on two processors, collecting default statistics
- mpsim.x -c2 -w — run on two processors, collecting power/energy statistics
- mpsim.x -c1 --is=9 --ds=10 — run on one processor with instruction cache of 512 bytes and data cache of 1024 bytes
- mpsim.x -c2 -F0,2 -F1,1 -F3,3 — run on two processors operating at 100 MHz and 200 MHz and the bus operating at 66 MHz
  - 200 MHz is the ”default” frequency
- mpsim.x -h — show other options
- Simulation results are in the file *stats.txt*

# Design-space Exploration

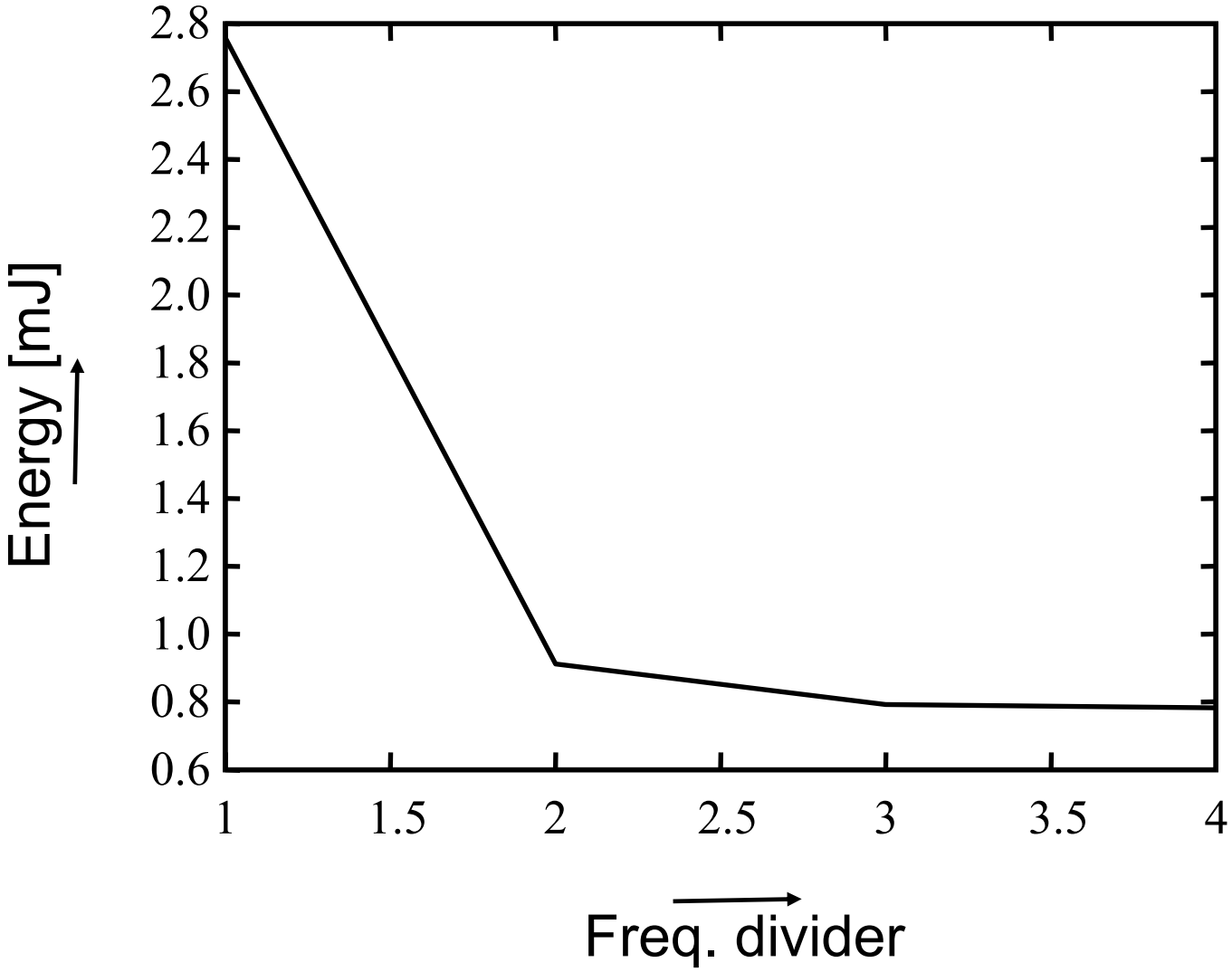
- Platform optimization
  - Select the number of processors
  - Select the speed of each processor
  - Select the type, associativity, and size of the cache
  - Select the bus type
- Application optimization
  - Select the interprocessor communication style (shared memory or distributed message passing)
  - Select the best mapping and schedule

# Energy/Speed Tradeoff

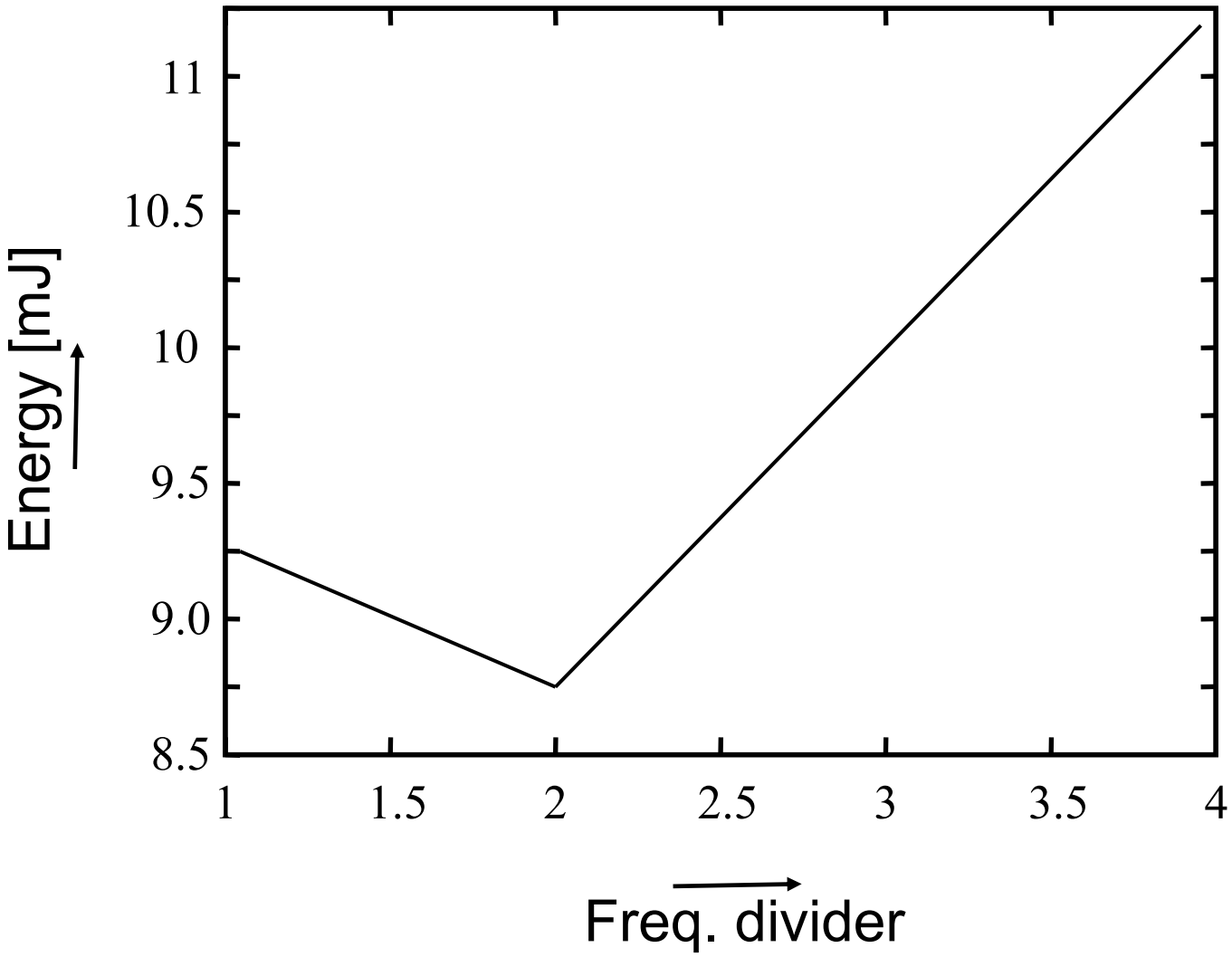
CPU model



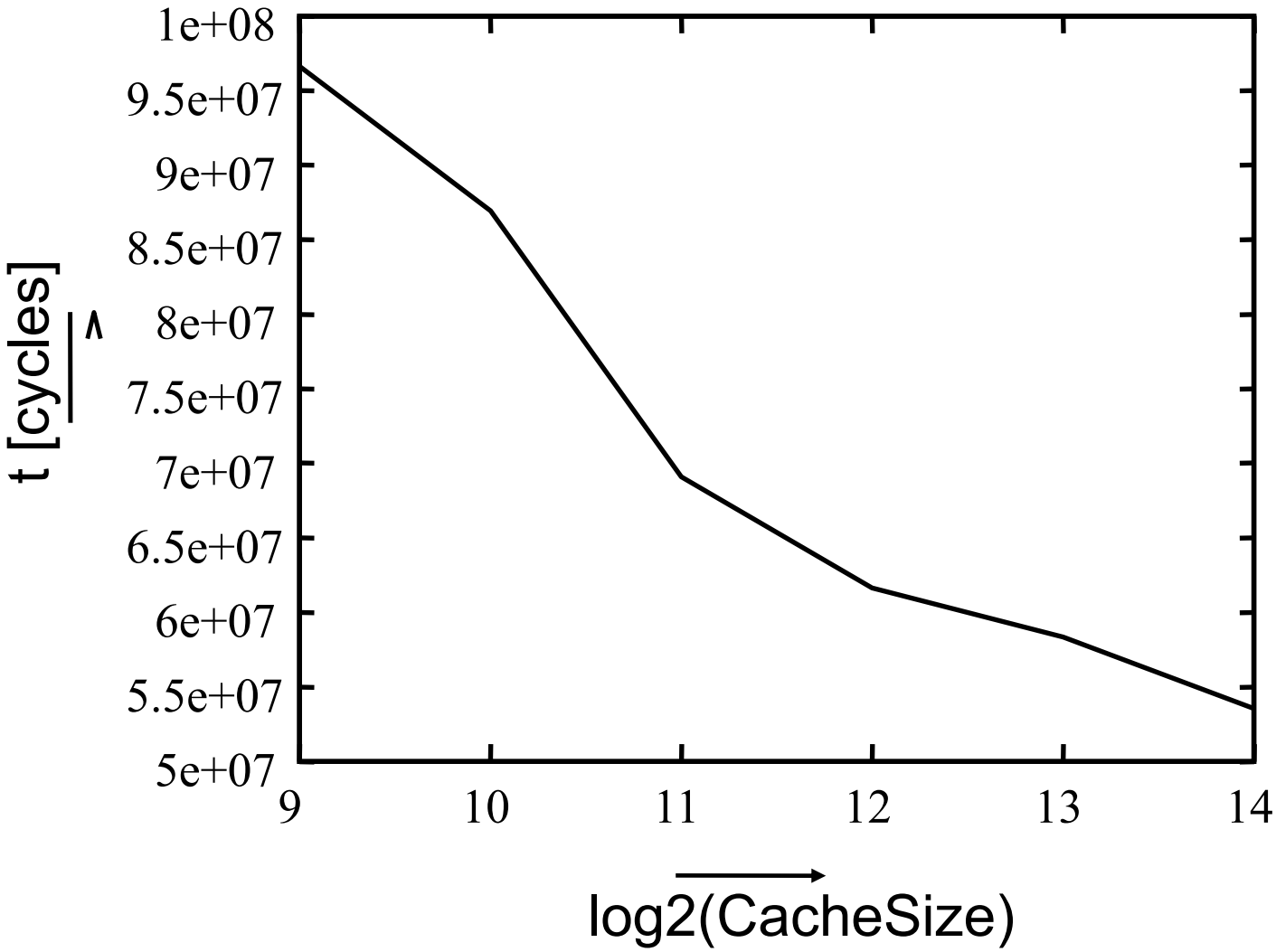
# Frequency Selection: ARM Core Energy



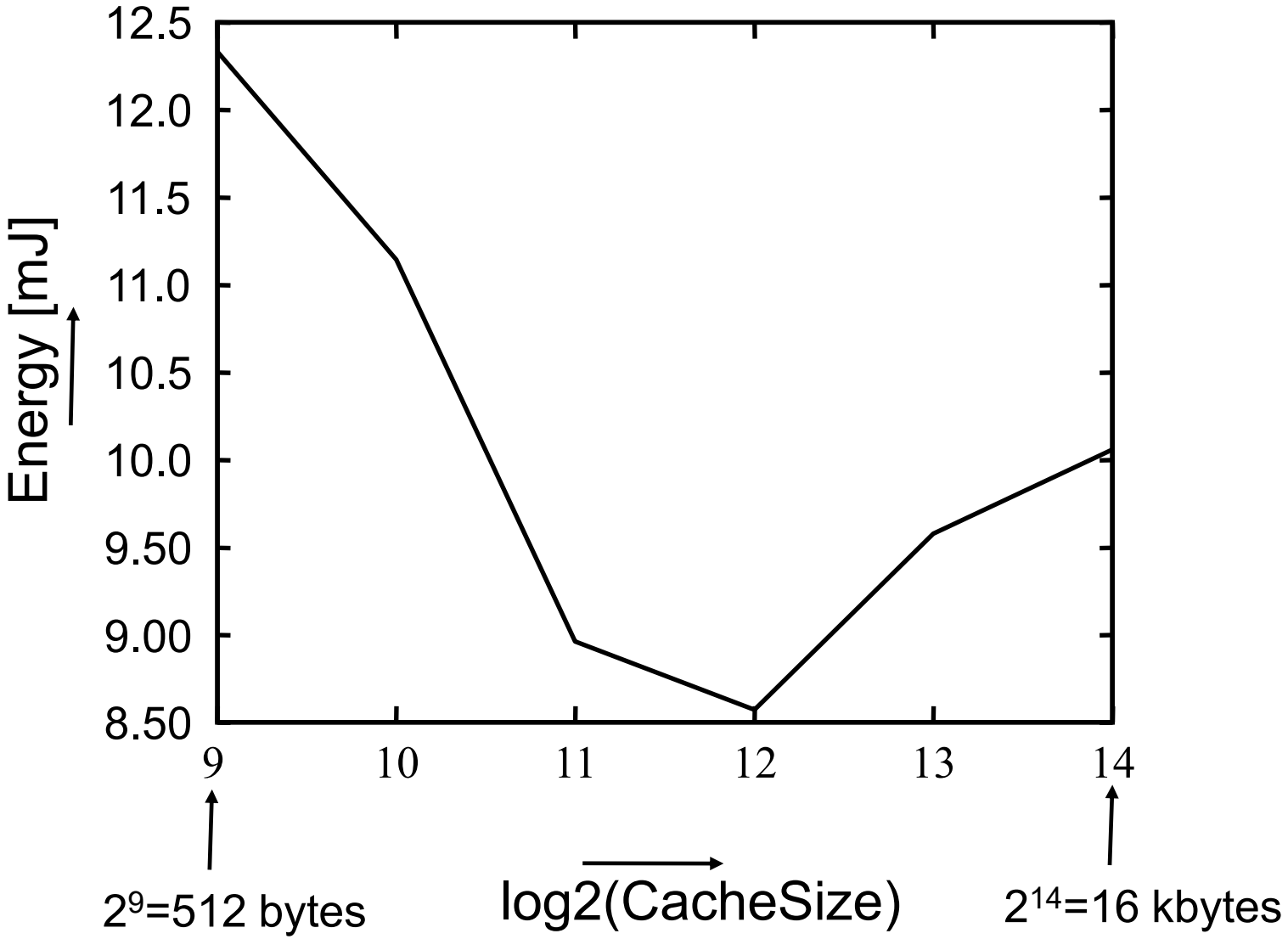
# Frequency Selection: Total Energy



# Instruction Cache Size: Execution Time



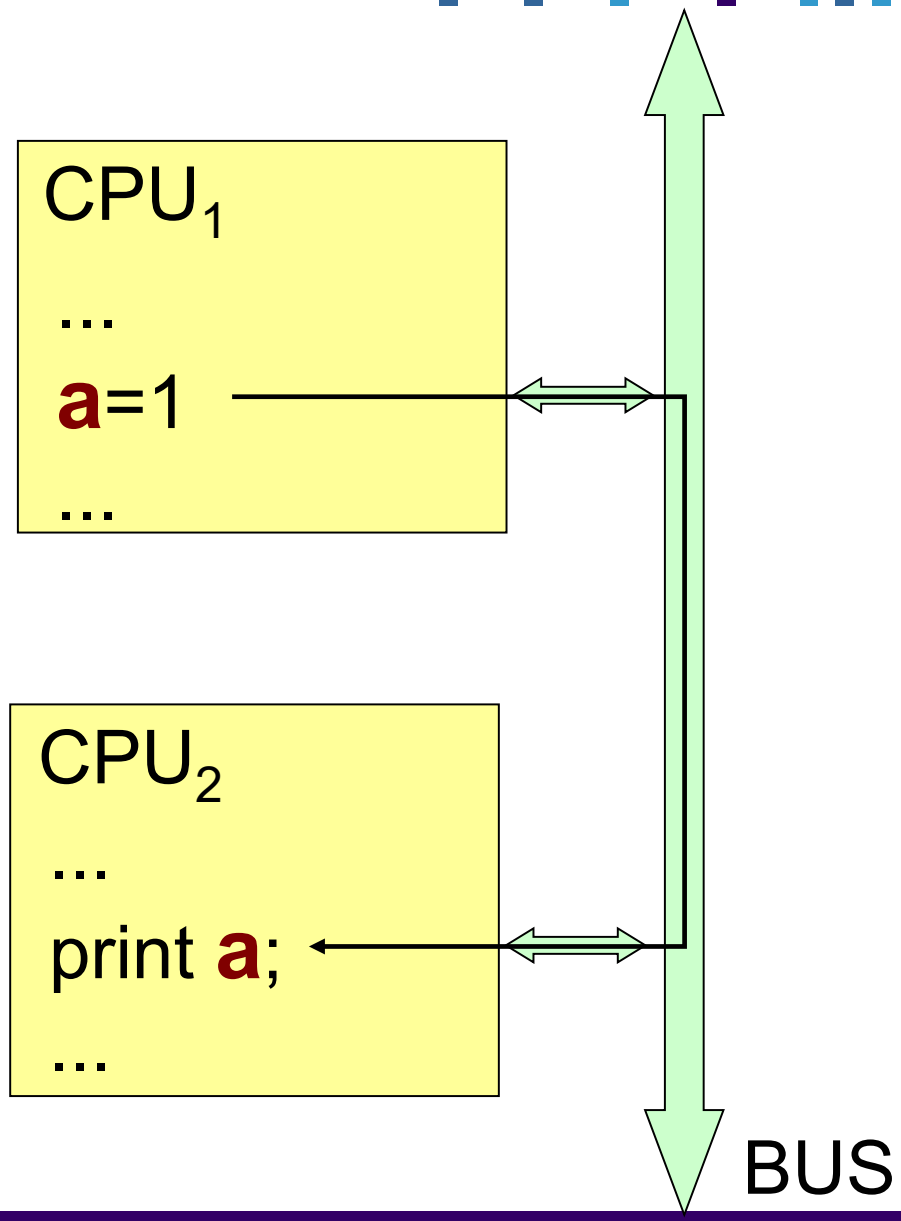
# Instruction Cache Size: Total Energy



# Assignment 2

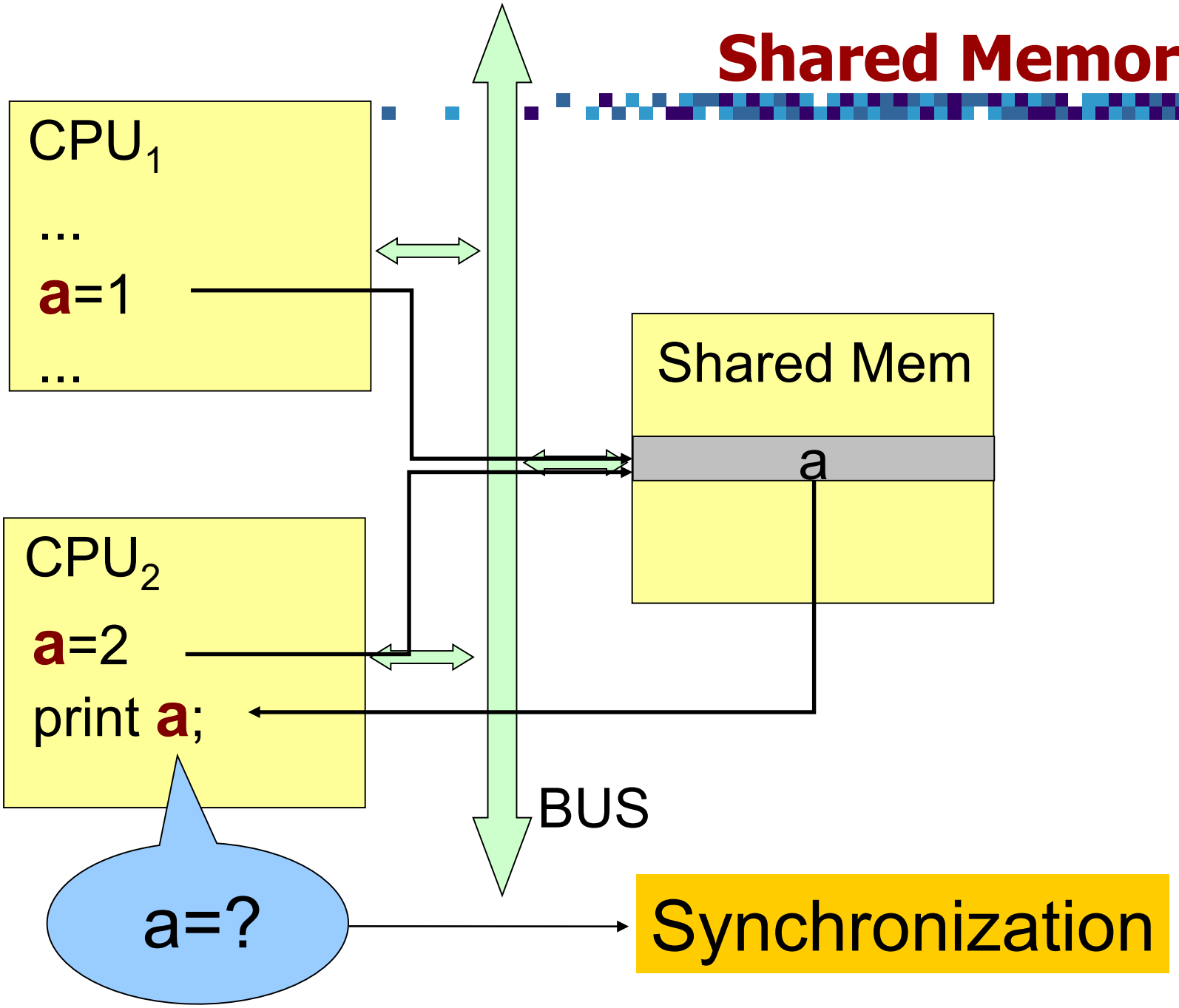
- Given a GSM codec
- Running on one ARM7 processor
- Variables
  - Cache parameters
  - Processor frequency
- Using MPARM, find a hardware configuration that minimizes the energy of the system

# Interprocessor Data Communication



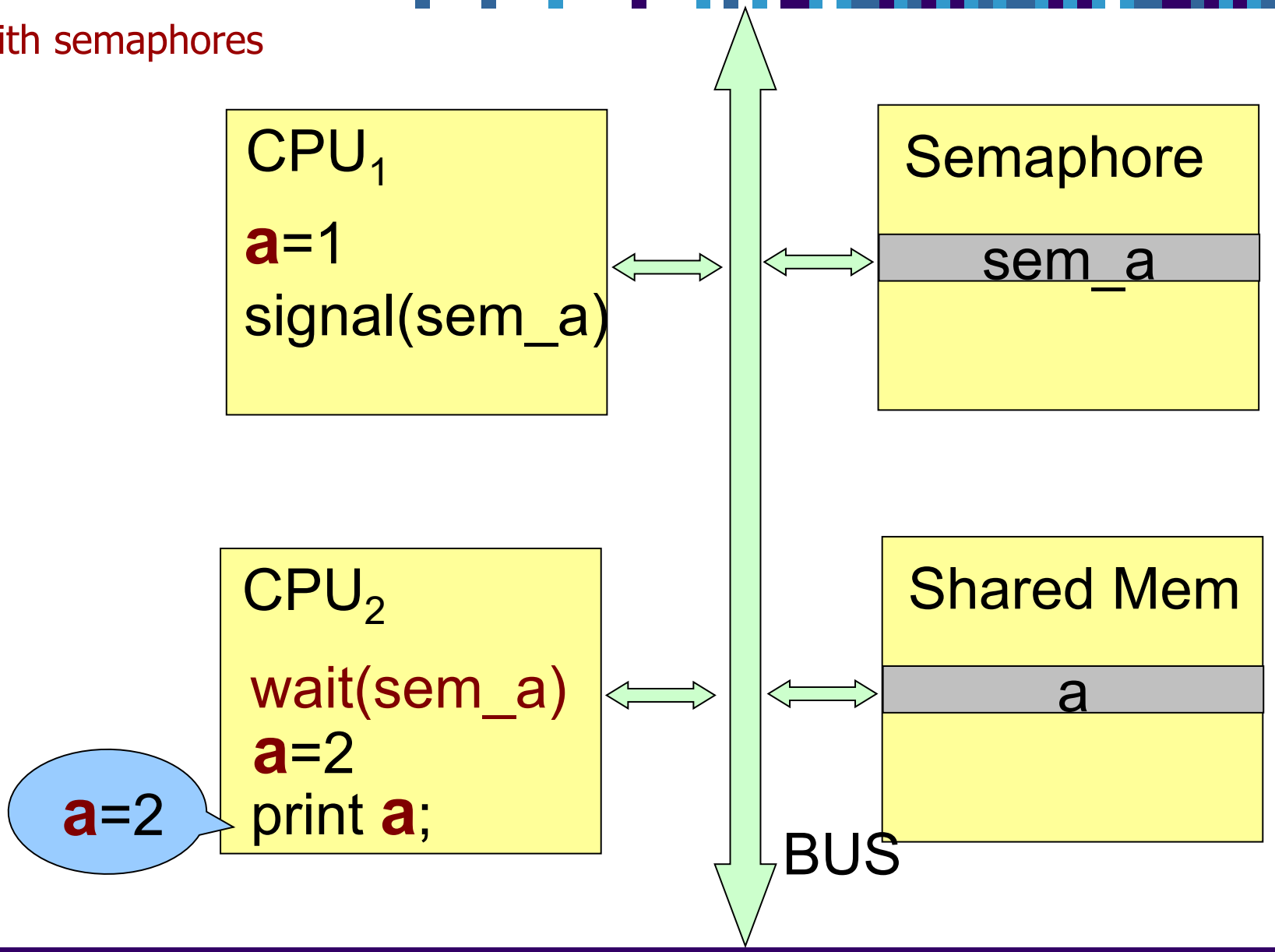

How?

# Shared Memory

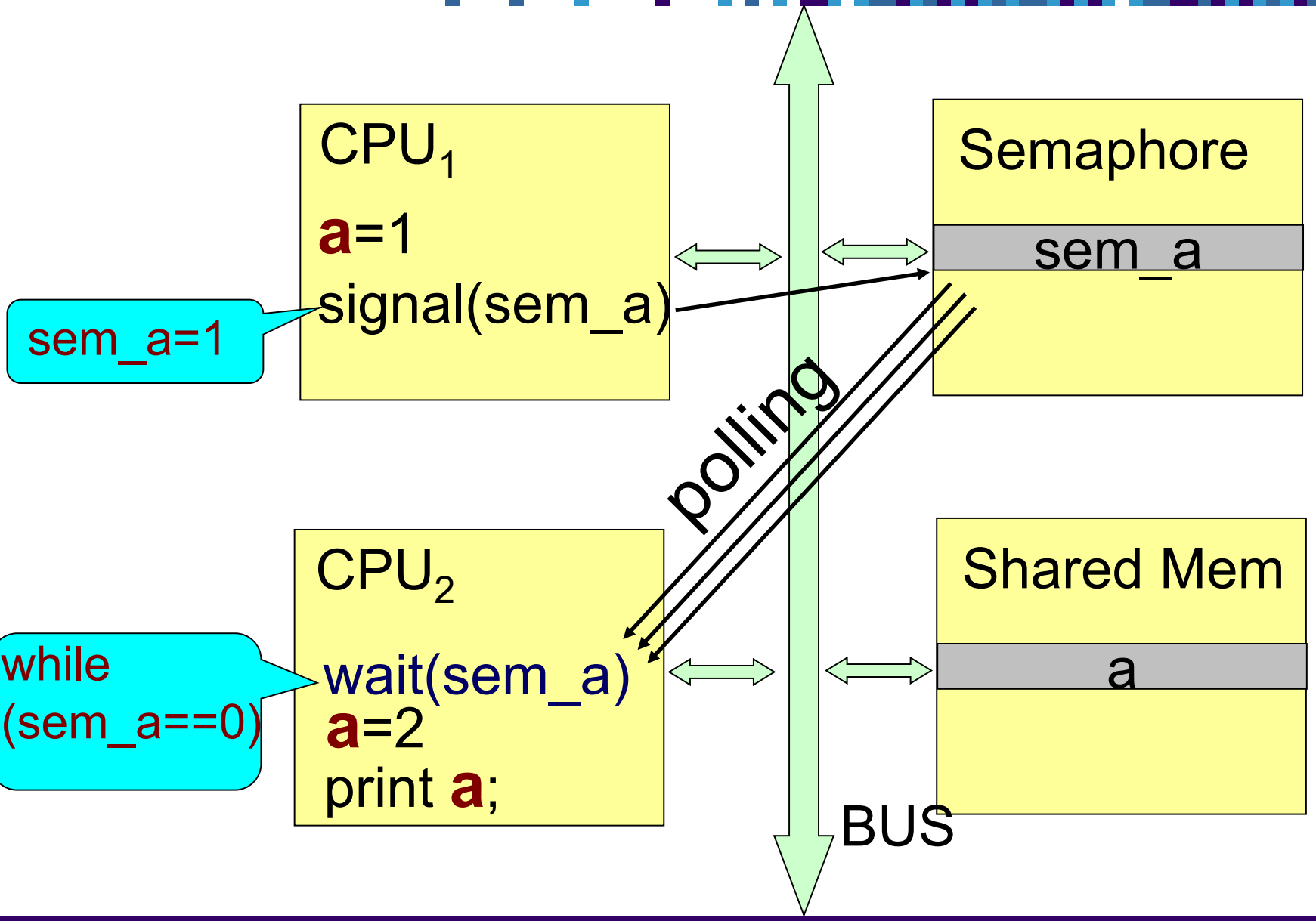


# Synchronization

With semaphores



# Synchronization Internals (1)



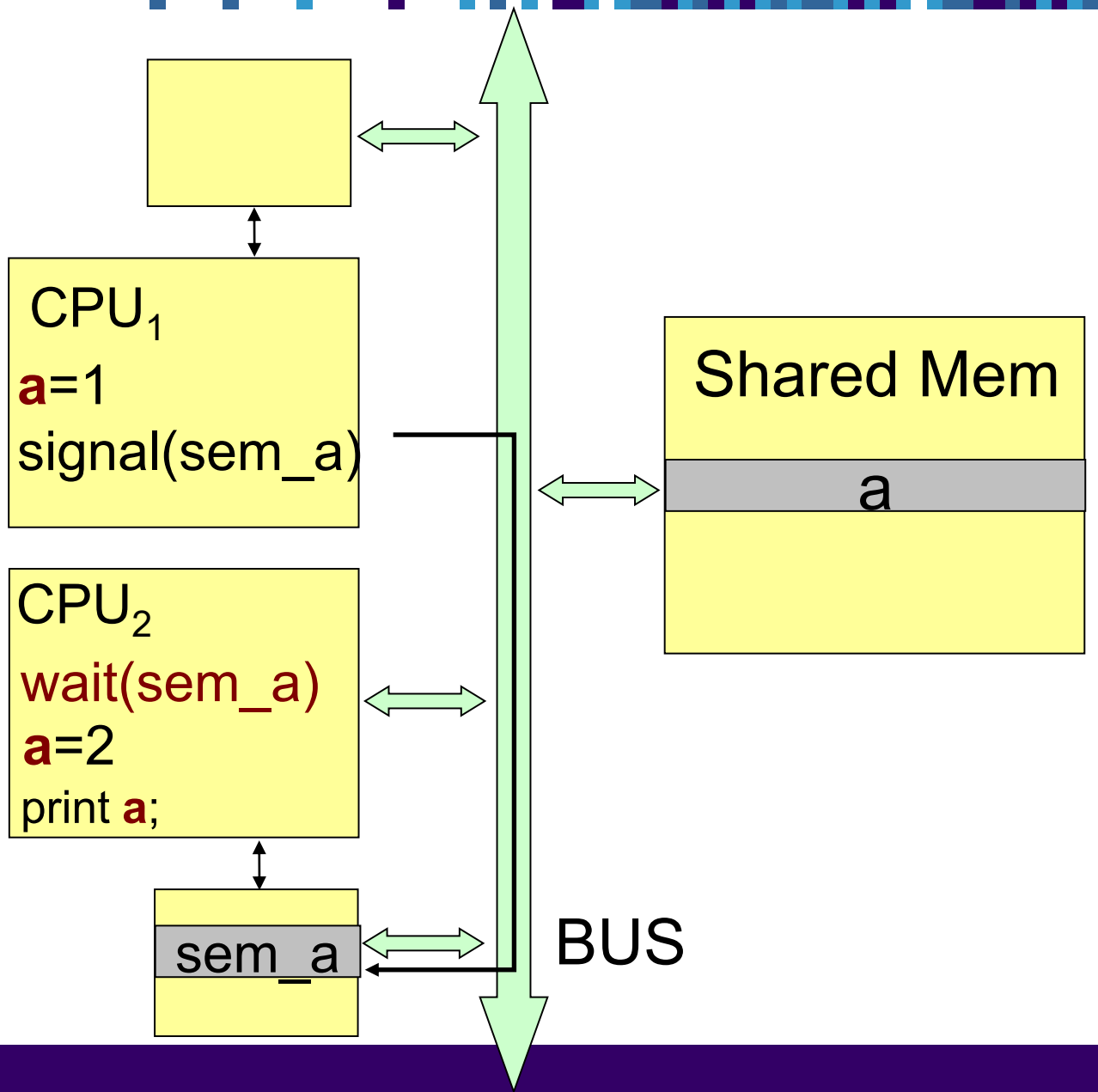
# Synchronization Internals (2)

- Disadvantages of polling
  - Results in higher power consumption
  - Larger execution time of the application
  - Blocking important communication on the bus

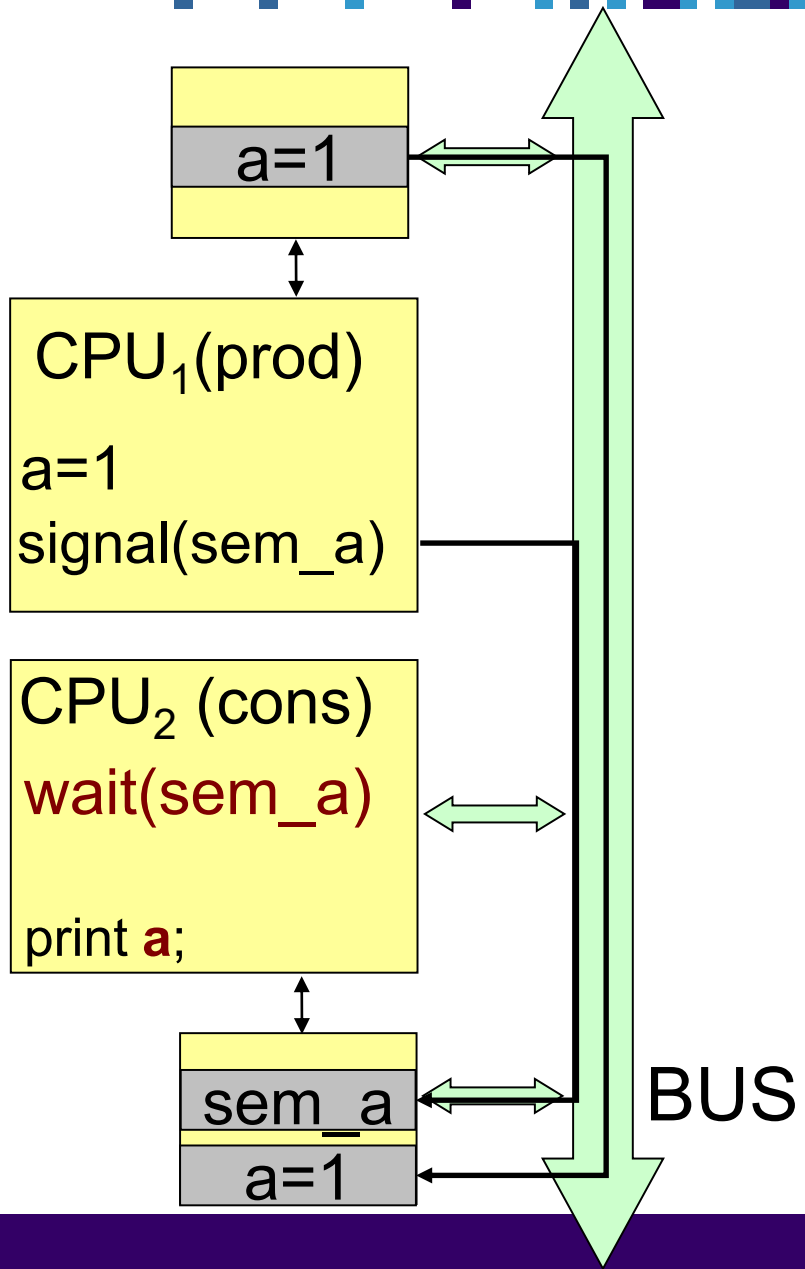
# Distributed Message Passing

- Direct CPU-CPU communication with distributed semaphores
- Each CPU has its own scratchpad
  - Smaller and faster than a RAM
  - Smaller energy consumption than a cache
  - Put frequently used variables on the scratchpad
  - Cache controlled by hardware
  - Scratchpad controlled by software
- Semaphores allocated on scratchpads
- No polling

# Distributed Message Passing (1)



# Distributed Message Passing (2)



# Assignment 2

- Given two implementations of the GSM codec
  - Shared memory
  - Distributed message passing
- Simulate and compare these two approaches
  - Energy
  - Runtime

**Thank you!**
**Questions?**