Mini Power-Aware TLM-Platform

Matthieu Moy

Verimag (Grenoble INP)
Grenoble
France

September 28th 2010

Summary

TLM-Power

- Library for power analysis on top of SystemC (CEA-LETI)
- Static and dynamic power-consumption computed as function of
  - Mode (idle, busy, . . .)
  - Phase (kind of computation performed)

TLM-Power: API

User code
`update_power(`*`mode`*`, `*`phase`*`)`
`set_static_power(`*`Watts`*`)` `set_dynamic_power(`*`Watts`*`)`

TLM-Power: mode/phase → power

- Mode/Phase: enumerated types
- Association (mode/phase → power) = text file:

```
# Configuration file of the consumer
#

ON          SEND            100mw     10mw
ON          ISEND           30mw      10mw
ON          SEND_ISEND      130mw     10mw
...
OFF                         0mw       0mw
```

TLM-Power: output

- Total power consumption =

$$\sum (P_s \cdot t + P_d \cdot t)$$

- Detailed trace
  - Text file
  - VCD file ($\rightarrow$ gtkwave & co)

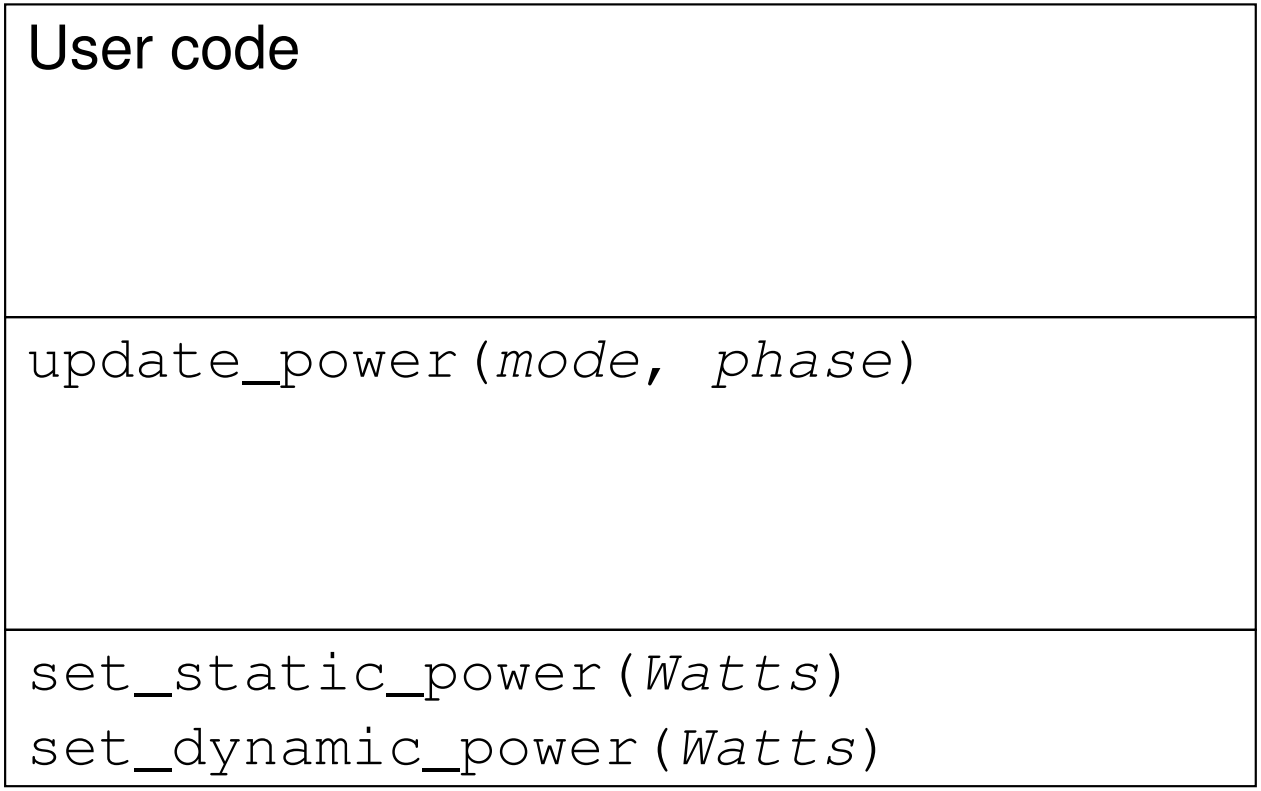
TLM-Power: the good and the bad

(hopefully no ugly)

- Good:
  - Existing API
  - Various output formats (VCD)
  - Exists and works already! ($\Rightarrow$ quick experiments)
- Bad:
  - Power depends on 2 parameters
  - Only enumerated types (cannot deal with continuous DVFS)
  - Text-file configuration limited

Summary

TLM-Power API, revisited



TLM-Power API, revisited

User code

```
virtual compute_static_power() = 0
virtual compute_dynamic_power() = 0
```

```
update_power()
set_value<type>("name", value)
get_value<type>("name")
```

```
set_static_power(Watts)
set_dynamic_power(Watts)
```

Benefits over plain TLM-Power

- Arbitrary number of parameters
- Arbitrary type for each parameter
- Arbitrary C++ code to compute static/dynamic power based on parameter value.

Example (1/2)

```
class my_power_params : public power_params {
public:
  my_power_params() {
    new_param<power_param_value<std::string> >("mode");
    new_param<power_param_value<int> >("bandwidth");
  }
  pw_power compute_static_power() {
    if (get_value<std::string>("mode") == "idle") {
      return pw_power(0, PW_WATT);
    } else if (get_value<std::string>("mode") == "turbo") {
      return pw_power(100, PW_WATT);
    } else {
      return pw_power(42 * get_value<int>("bandwidth"), PW_WATT);
    }
  }

  pw_power compute_dynamic_power() {
    return pw_power(72, PW_WATT);
  }
};
```

Example (2/2)

```
class hello : public sc_module,
              public pw_param_module<my_power_params> {
  void compute(void) {
    while (true) {
      set_value<int>("bandwidth", get_value<int>("bandwidth") + 1)
      set_value<std::string>("mode", "idle");
      update_power();
      wait(1, SC_SEC);
      set_value<std::string>("mode", "run");
      update_power();
      wait(1, SC_SEC);
    }
  }
  SC_HAS_PROCESS(hello);
public:
  hello(sc_module_name name) : sc_module(name),
    pw_param_module<my_power_params>() {
    SC_THREAD(compute);
  }
};
```

Pointers to code

- Example: `tlm-power-cea/test/moy/hello/sc_main.cpp`
- Library code: `tlm-power-cea/include/pw_param.h`, `tlm-power-cea/src/pw_param.cpp` (re-uses `pw_module_base` but not `pw_module`)

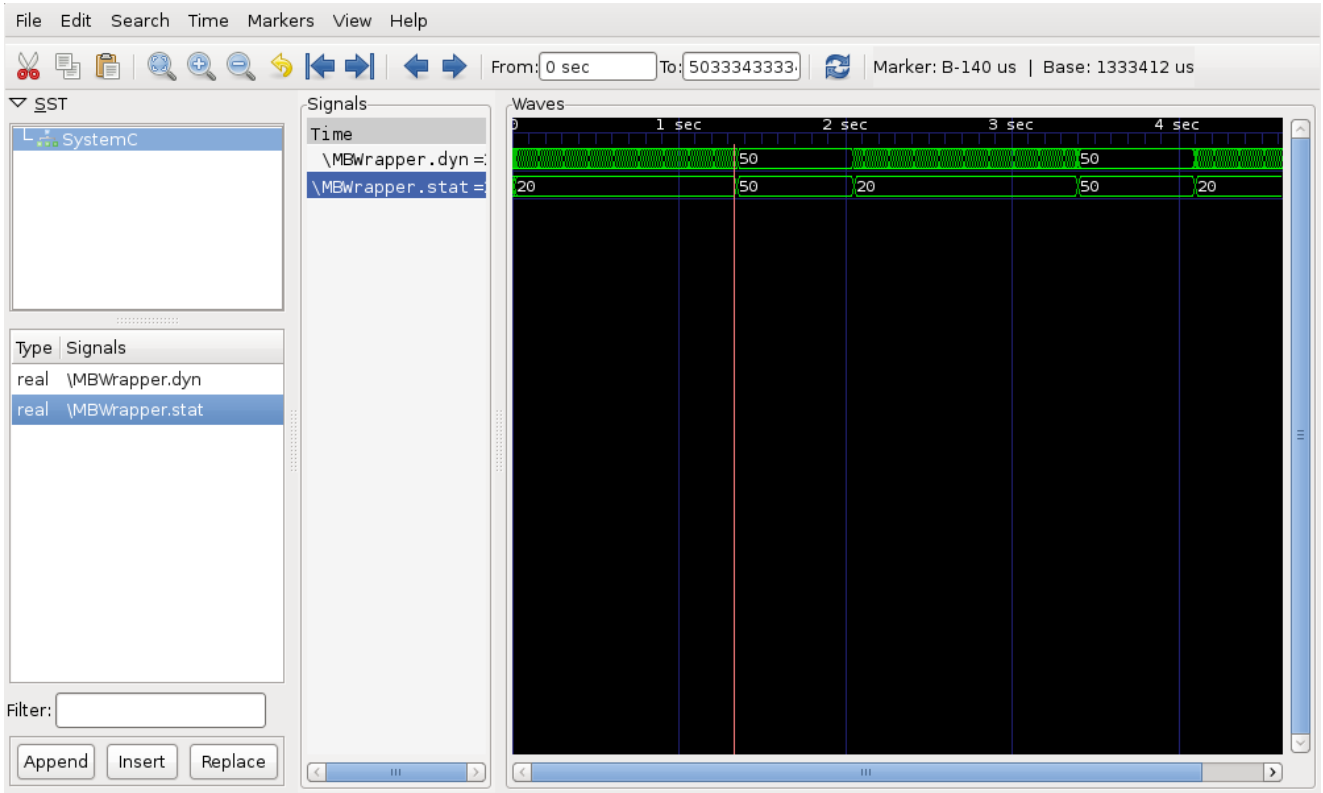
Summary

The platform

- Based on a small but real (i.e. runs on FPGA) platform
- Power management and estimation added artificially
- Focus on CPU (implemented with ISS)
- Toy implementation for:
  - Frequency scaling: frequency becomes a parameter
  - `sleep` instruction: one “mode” parameter (sleep/run)

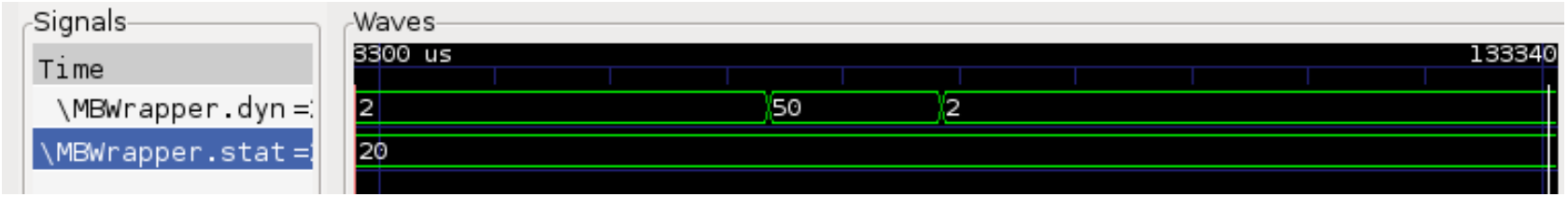
Example execution

- Compute next image (game of life)
- Wait for LCDC interrupt
- Switch display to next image (double-buffering)
- Wait for timer interrupt (long $\Rightarrow$ can reduce frequency)

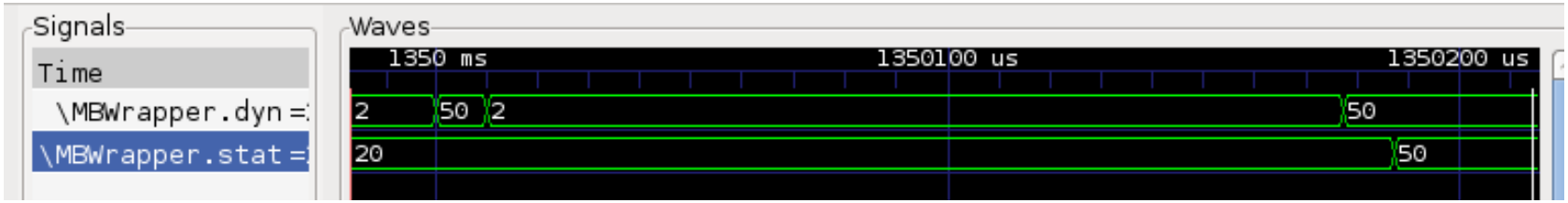


Example execution: details

- LCDC Interrupt:
  - Wake up to execute IRQ routine
  - Remain at low frequency



- Timer Interrupt:
  - Increase frequency
  - Start computing



DVFS: Implementation (1/2)

- New component: Power-controler
  - 1 control register: written value = new frequency
  - Frequency sent to CPU through `sc_signal<int>`
- Embedded software:

```
void game_of_life() {
  write_mem(POWER_CONTROLER_BASEADDR
              + POWER_CONTROLER_CPUFREQ,
            40); // Go fast
  for(x = 0; x < SW_VGA_WIDTH; ++x) {
    // CPU-intensive computation
  }
  write_mem(POWER_CONTROLER_BASEADDR
              + POWER_CONTROLER_CPUFREQ,
            10); // Go slow
}
```

DVFS: Implementation (2/2)

- CPU (constraint: avoid being intrusive)

```
SC_CTOR(MBWrapper) {
  SC_METHOD(freq_change);
  sensitive << freq;
}

void MBWrapper::freq_change() {
  // Power
  set_value<int>("freq", freq.read());
  update_power();
  // Timing
  m_period = sc_core::sc_time
    (1000/freq.read(), sc_core::SC_NS);
}
```

Instruction `sleep`: implementation (HW)

- Processor (MicroBlaze) doesn't have an instruction to wait for interrupt
- $\Rightarrow$ add one (did I already mention "artificial"?)

```
while(true) {
  if (m_iss.m_sleeping) {
    set_value<std::string>("mode", "sleep");
    update_power();
    wait(irq.posedge_event());
    m_iss.m_sleeping = false;
    set_value<std::string>("mode", "run");
    update_power();
  }
  switch(opcode) {
  ...
  case OP_SLEEP:
    m_sleeping = true;
  }
}
```

Instruction `sleep`: implementation (SW)

```
/* Fake "sleep" instruction. */
#define wait_for_irq()  \
        __asm(".byte 0x50, 0x0, 0x0, 0x0")
```

Summary

Conclusion

- We have:
  - Toy implementation for DVFS and modes in ISS (HW+SW)
  - Basic integrator ($E = P \cdot t$)
  - An example of layered API
- We don't have:
  - Connection to an advanced solver (Docea...)
  - Anything realistic
  - Power-managed components other than CPU
- We could:
  - Reuse the `set_param()`/`get_param()` API
  - Do something more clever in the back-end
  - *Cornet*ify the approach (PV+T+E)