Effective Timekeeping in RISC-V Processors

Overview of mtime Design Approaches

Viktor J. Schneider (IFAG DES PTS TI EA)

September 15, 2023

Agenda

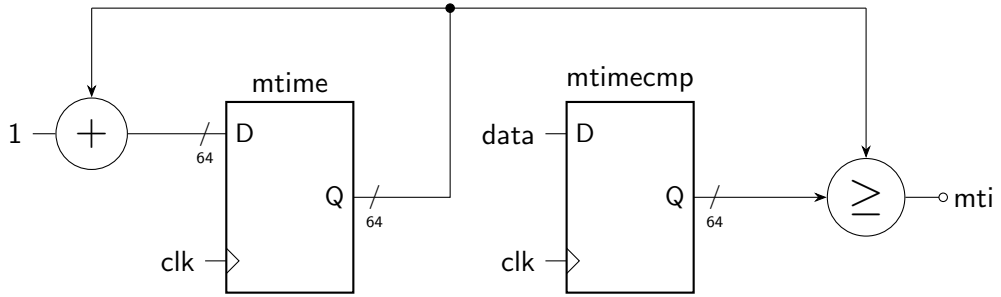
1. What is mtime?
2. What makes mtime special?
3. Design approaches
4. Different approach for small embedded cores

What is mtime?

- 64 bit timer defined in RISC-V specification
- Provides wall-clock time
- Can raise interrupts at specific time points
- Necessary for
  - Scheduling in an OS
  - Triggering tasks at specific time points

RISC-V mtime

- Provides stable wall-clock Time
- One of RISC-V's hardware performance-monitoring counters
- Consists of two registers: *mtime* and *mtimecmp*



Hardware Performance Monitor

- Consists of 32 hardware performance-monitoring counters:
  - *mcycle*
    Counts clock cycles
  - *minstret*
    Counts retired instructions
  - *mtime*
    Counts wall-clock time
  - *mhpmcounter3-31*
    Count custom events
- Access is generally only allowed for machine mode
- Shadow CSRs allow access to less privileged modes

Agenda

1. What is mtime?
2. What makes mtime special?
3. Design approaches
4. Different approach for small embedded cores

What makes mtime special?

- makes use of **memory-mapped** Registers
  - Only module with memory-mapped Registers in RISC-V spec.
  - Access through memory bus
- Less privileged modes can access *mtime* through the **shadow CSR** *time*
- How is underlying data accessed for this shadow CSR?
  - CSR accesses use special instructions (e.g. CSRRW)
  - Memory accesses use LOAD and STORE instructions

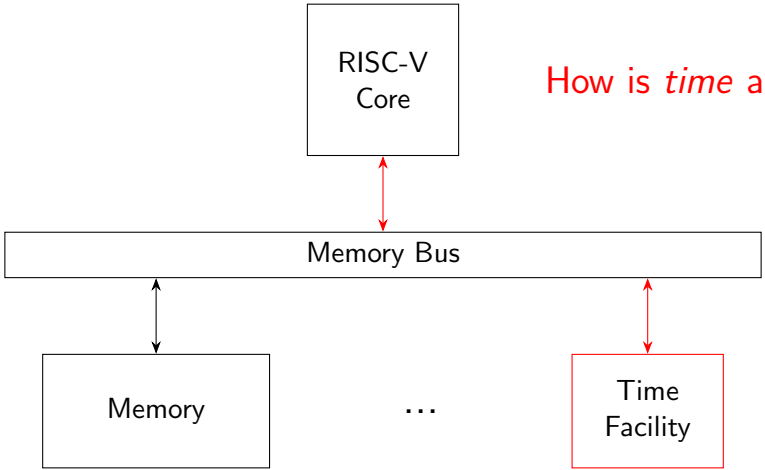
Why memory-mapped registers?

- Timer might be implemented outside the core:
  - Located in special always-on, low-power domain
  - Separate module for driving a crystal or oscillator
  - Shared between multiple cores
  - Dynamic frequency scaling makes core clock unusable

Agenda

1. What is mtime?
2. What makes mtime special?
3. Design approaches
4. Different approach for small embedded cores

How to implement mtime?



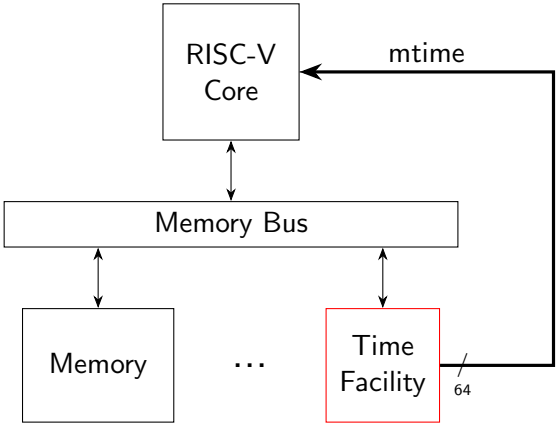
Emulating behavior in software

- Not implementing time (in hardware)
- What happens when accessing time?
  1. CSRRW targeting *time* CSR
  2. Illegal Instruction fault is raised
  3. Trap handler is called (in Machine Mode)
  4. Trap handler reads mtime and emulates the time access
- Complexity is moved to software side
- Easy to implement in hardware
- Slow access time
- This approach is used in e.g. *IBEX* and *NEORV32*

Connecting mtime to the core

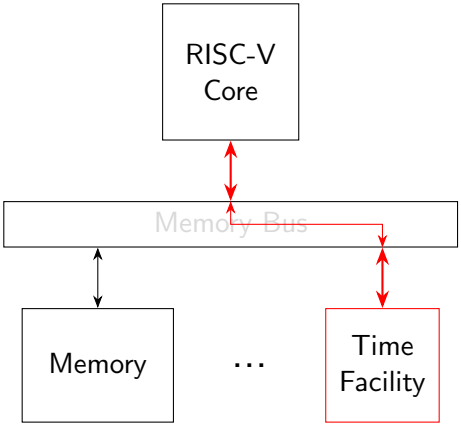
- 64 bit line necessary
- Fast hardware access

- Approach can be found in *Bluespec Flute*

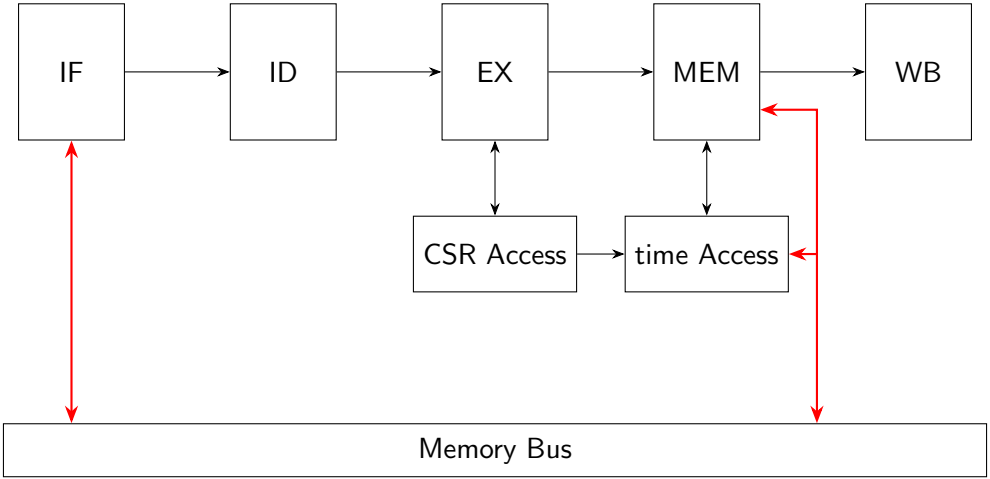


Using the memory bus

- Allow CSR instructions to access memory bus
- Reduces area usage
- May introduce stalls



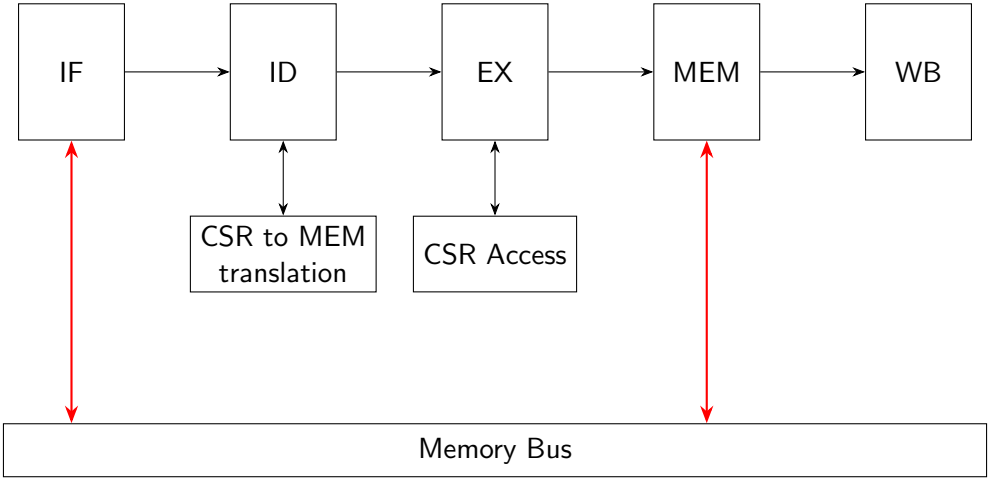
Using the memory bus



Instruction translation

- Special handling of time CSR instructions
- Translate CSR instructions to LOAD ~~and STORE~~ instructions
- Challenges:
  - ~~Issue two instructions per CSR access (LOAD and STORE)~~
  - ~~Needs to be atomic~~
- STORE not necessary because *time* is a read-only CSR

Instruction translation



Agenda

1. What is mtime?
2. What makes mtime special?
3. Design approaches
4. Different approach for small embedded cores

A different approach

- What about small (deeply) embedded cores?
  - No dynamic frequency scaling
  - Single clock domain
  - Single core
- CSRs instead of memory-mapped registers
  - Reduces complexity
  - Consistent with other HPM counters
  - Faster access time

Reducing area usage

- mtime and mcycle use the same clock
- Combine mtime and mcycle counter:
  - Remove *mtime*
  - Provide *time* as a shadow of *mcycle*
  - Compare *mtimecmp* with *time*
- Only machine mode software is affected
- A similar approach can be found in *Rocket Chip*
  - *time* is shadow of *mcycle*

Summary

- mtime uses memory-mapped registers
- Shadow CSRs of such registers are non-trivial to implement
- Embedded cores can benefit from a pure CSR design
  - Consistent with other HPM counters
  - Faster access time
  - Less area usage