Multi-core RISC Processor Design & Implementation

Demonstration Viva

Ben Lancaster

201280376
ELEC5881M - Main Project

July 7, 2019

Quick Links

- GitHub repository: `https://github.com/bendl/vmicro16`
- Full Report: `https://github.com/bendl/vmicro16/blob/master/docs/reports/build/ELEC5881M_Ben_Lancaster_201280376_Final.pdf`

Why Multi-core?

Block Title

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Why RISC?

Overview

What this project produces:

- **System-on-Chip with multi-processor functionality**
  Tested on FPGA hardware with 1-96 CPU cores.

Overview

What this project produces:

- **System-on-Chip with multi-processor functionality**
  Tested on FPGA hardware with 1-96 CPU cores.

- **Custom 16-bit RISC CPU**
  With interrupts and its own Instruction Set Architecture (ISA).

Overview

What this project produces:

- **System-on-Chip with multi-processor functionality**
  Tested on FPGA hardware with 1-96 CPU cores.

- **Custom 16-bit RISC CPU**
  With interrupts and its own Instruction Set Architecture (ISA).

- **Aimed at Design Engineers, not end users**
  Project is provided as source code/design files for Design Engineers to customise and implement in hardware themselves.

Overview

What this project produces:

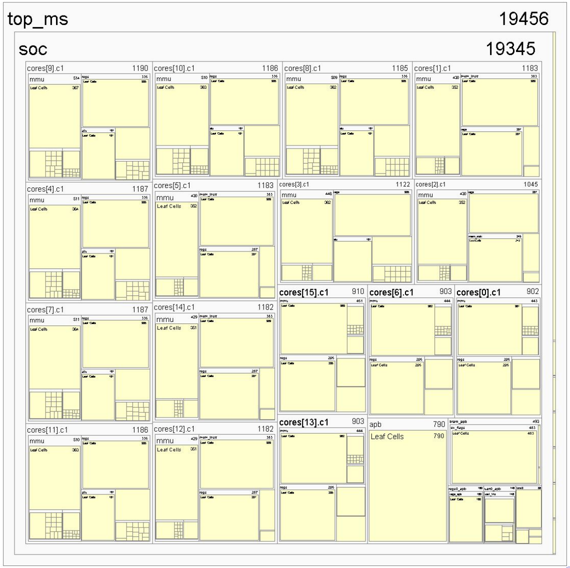
- **System-on-Chip with multi-processor functionality**
  Tested on FPGA hardware with 1-96 CPU cores.

- **Custom 16-bit RISC CPU**
  With interrupts and its own Instruction Set Architecture (ISA).

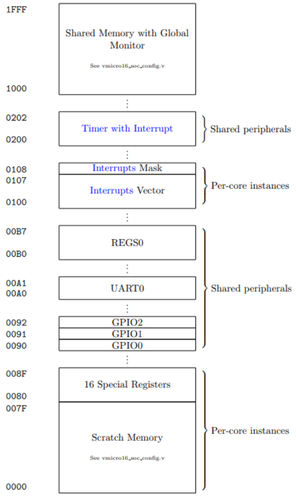
- **Aimed at Design Engineers, not end users**
  Project is provided as source code/design files for Design Engineers to customise and implement in hardware themselves.

- **Software/Assembly compiler**
  PRCO304 programming language/Intel assembly syntax.

Top Level Hierarchy



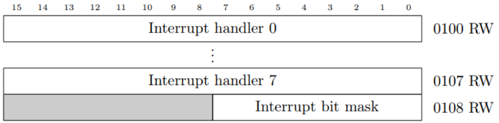
Memory Map

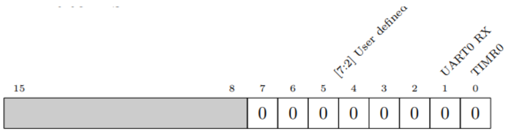


- **Shared Memory with Global Monitor**
- Timer with Interrupt
- Per-core Interrupt Vector and Mask
- Shared Register Set
- UART Transceiver
- Multiple GPIO ports
- Per-core scratch memory
- **Per-core Special Registers**
- Customisable by designers

Interconnect

Interrupts





```
entry:
    // Set interrupt vector at 0x100
    // Move address of isr0 function to vector[0]
    movi    r0, isr0
    // create 0x100 value by left shifting 1 8 bits
    movi    r1, #0x1
    movi    r2, #0x8
    lshft   r1, r2
    // write isr0 address to vector[0]
    sw      r0, r1

    // enable all interrupts by writing 0x0f to 0x108
    movi    r0, #0x0f
    sw      r0, r1 + #0x8
    halt                    // enter low power idle state

isr0:                       // arbitrary name
    movi    r0, #0xff       // do something
    intr                    // return from interrupt
```

Demo: 2 Core LED toggle (GPIO0) with TIMR0 1s interrupt (interrupts_2.s)

Timer Interrupt Example

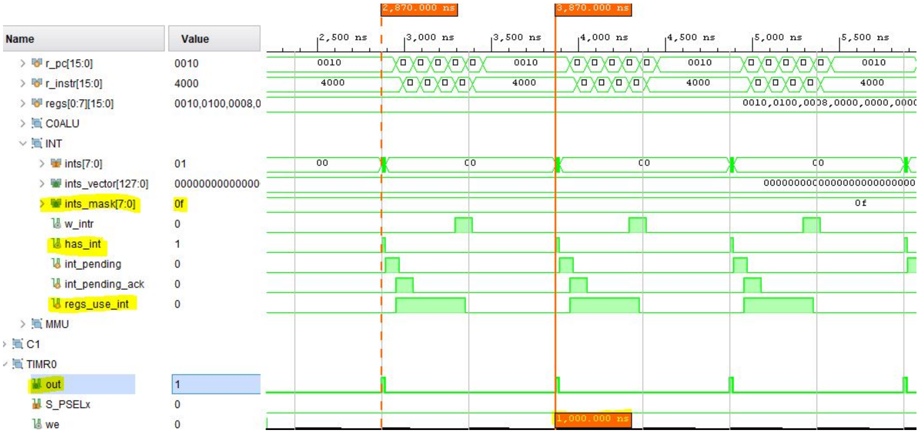


Figure: TIMR0 1us interrupt with context switching

Timer Peripheral Registers

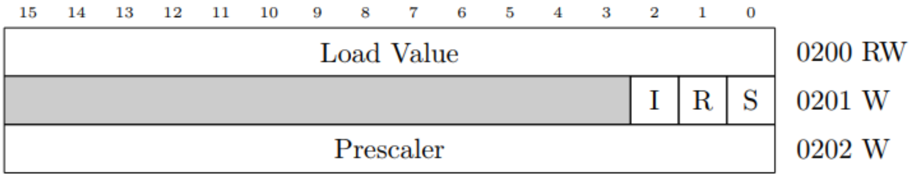


Figure: $t = 20ns * load * prescaler$

Resolution (32-bit timer): 20ns to 85s.
Examples:

- For 1us: Load = 0x32, Prescaler = 0 (20ns * 0x32 = 1000ns)
- For 1s: Load = 0x1000, Prescaler = 0x3000 (demo) (20ns * 0x1000 * 0x3000 = approx. 1s)

HW/SW Requirements

Hardware:

- **Bus Arbitration**
  (scheduling: priority, rotating, etc.)

Software:

HW/SW Requirements

Hardware:

- **Bus Arbitration**
  (scheduling: priority, rotating, etc.)
- **Atomic functions**
  (atomic versions of load/store to prevent race conditions)

Software:

HW/SW Requirements

Hardware:

- **Bus Arbitration**
  (scheduling: priority, rotating, etc.)
- **Atomic functions**
  (atomic versions of load/store to prevent race conditions)
- Per-core instruction memory

Software:

HW/SW Requirements

Hardware:

- **Bus Arbitration**
  (scheduling: priority, rotating, etc.)
- **Atomic functions**
  (atomic versions of load/store to prevent race conditions)
- Per-core instruction memory
- Per-core context-switching for interrupt handling

Software:

HW/SW Requirements

Hardware:

- **Bus Arbitration**
  (scheduling: priority, rotating, etc.)
- **Atomic functions**
  (atomic versions of load/store to prevent race conditions)
- Per-core instruction memory
- Per-core context-switching for interrupt handling

Software:

- Semaphores/Mutexes
  (exclusive memory access)

HW/SW Requirements

Hardware:

- **Bus Arbitration** (scheduling: priority, rotating, etc.)
- **Atomic functions** (atomic versions of load/store to prevent race conditions)
- Per-core instruction memory
- Per-core context-switching for interrupt handling

Software:

- Semaphores/Mutexes (exclusive memory access)
- Thread synchronisation (memory barriers)

HW/SW Requirements

Hardware:

- **Bus Arbitration**
  (scheduling: priority, rotating, etc.)
- **Atomic functions**
  (atomic versions of load/store to prevent race conditions)
- Per-core instruction memory
- Per-core context-switching for interrupt handling

Software:

- Semaphores/Mutexes
  (exclusive memory access)
- Thread synchronisation
  (memory barriers)
- **Context identification**
  What core am I?
  How many cores?
  How much memory?

Context Identification

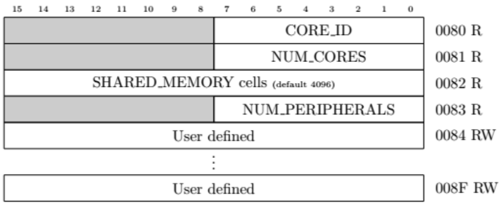


Figure: Special Registers 0x0080 to 0x008F

```
entry:
    // get core idx 0x80 in r7
    movi    r7, #0x80
    lw      r7, r7

    // Branch away if not core 0
    cmp     r7, r0
    movi    r0, exit
    br      r0, BR_NE

    // Core 0 only instructions
    nop     r0, r0
    nop     r0, r0
    nop     r0, r0

exit:
    halt    r0, r0
```

Results

Results 1

Accomplishments

- **Near complete System-on-Chip design with various peripherals**
  Timers, GPIO, UART, Registers, Memory
- **Common multi-thread/core synchronisation primitives**
  Semaphores, Mutexes, Memory Barriers, Atomic Instructions
- **AMBA APB bus interface with Global Monitor**
  Timers, GPIO, UART, Registers, Memory
- **Working shared bus arbitration**
  Schedules access to shared resources
- **Working FPGA implementation for a 96 core design**
  Nearly fills Cyclone V FPGA on the DE1-SoC
- **Interrupts with hardware context-switching**
  Low latency to react to interrupt
- **Acknowledges design limitations and attempts to overcome**
  LUT resources, block memories, power and temperature requirements

Future Improvements

- **Working Global Reset**
  Global resets are expensive (LUT resources)
  Resetting block memories is not trivial
- **On-chip Programming**
  Use the UART0 receiver to program each cores flash memory
- **Per-core gating/enabling**
  Improve power efficiency for ASIC implementation by disabling cores at run-time via software.
- **Improve memory bottleneck**
  Each core requires it's own memory - reduce by multiplexing access to a single large memory.

Q&A