Multi-core RISC SoC Design & Implementation

Demonstration Viva

Ben Lancaster

201280376
ELEC5881M - Main Project

July 22, 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`
- This presentation: `https://github.com/bendl/vmicro16/blob/master/docs/reports/build/ELEC5881M_Ben_Lancaster_201280376_viva.pdf`
- About me: `https://bendl.me/`

Why a project on CPUs?

- **CPUs will be used for the rest of humanity**
  1000s of years
- **Understand design constraints and considerations**
  Be a better engineer
- **Prepare myself for future employment/work**

Why Multi-core?

- **Rate of single-core speed improvements slowing**
  Pipelining, register renaming, branch predictions, OoOE, clock speeds
- **Future of computing = parallel**
  - Identifying parallel opportunities
  - Massively parallel (NoC's)
  - Higher throughput

Why RISC?

- Easier design & impl
- Smaller = fit more cores on a chip
- FPGA size limitations
- Previous experience + future work
- I’m a RISC purist

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

Overview

What this project produces:

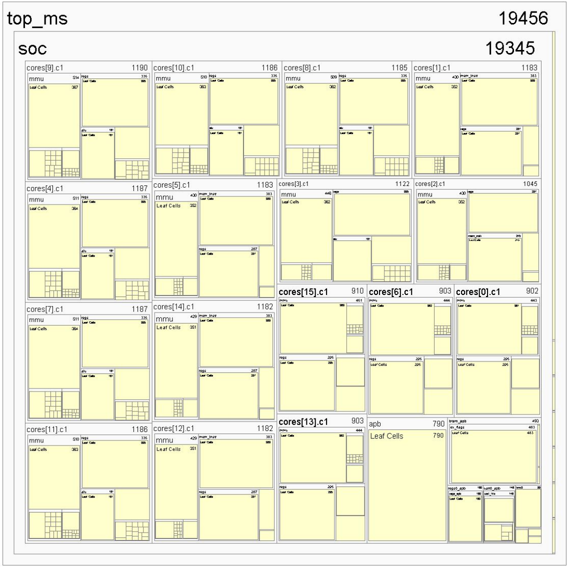
- **System-on-Chip with multi-processor functionality**
  Tested on FPGA hardware with 1-96 CPU cores.

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

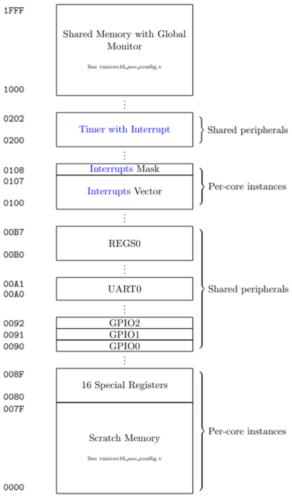
- **Software/Assembly compiler**
  PRCO304 programming language/Intel assembly syntax.

- **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.

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

- AMBA APB Bus
- Tristate & Non-tristate (mux) impl
- Originally Wishbone, now APB
- AHB too complex (limited time)
- Various schedulers

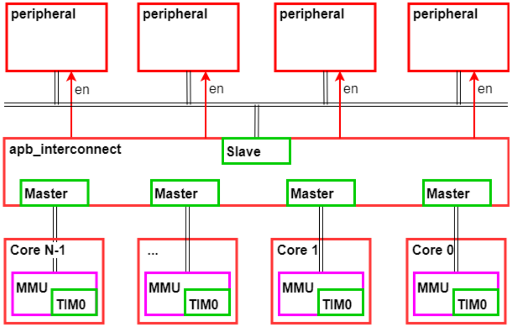


Figure: Vmicro16 interconnect

Interconnect Schematic

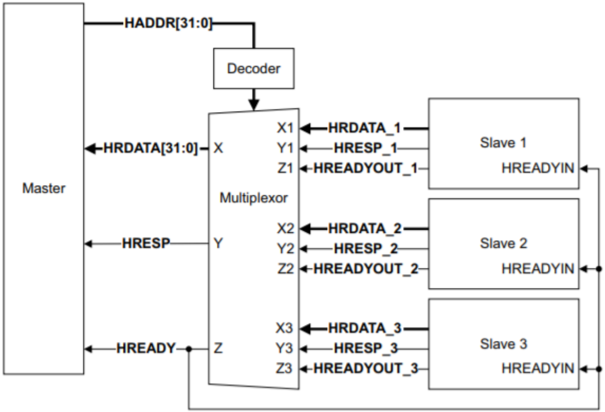


Figure: Source: ARM AHB-Lite Protocol Specification Figure 4-2.

Instruction Set Architecture

+ 16-bits, 38 instructions
+ Simple load/store arch
+ (Un)signed instructions
+ Compact
+ Optional hardware multiply instruction

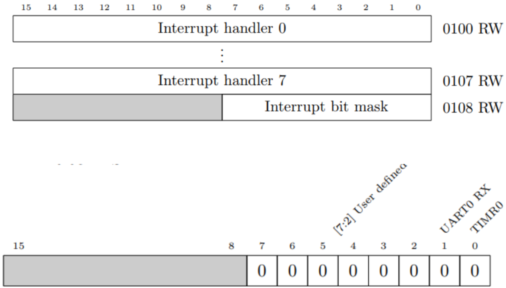
— Only 8 registers

— No good compiler support

	15-11	**10-8**	**7-5**	**4-0**	**rd ra simm5**
	15-11	**10-8**	**7-0**		**rd imm8**
	15-11	**10-0**			**nop**
	15	**14:12**	**11:0**		**extended immediate**
SPCL	**00000**	**11 bits**			**NOP**
SPCL	00000	11h'000			NOP
SPCL	00000	11h'001			HALT
SPCL	00000	11h'002			Return from interrupt
LW	00001	Rd	Ra	s5	Rd <= RAM[Ra+s5]
SW	00010	Rd	Ra	s5	RAM[Ra+s5] <= Rd
BIT	**00011**	**Rd**	**Ra**	**s5**	**bitwise operations**
BIT_OR	00011	Rd	Ra	00000	Rd <= Rd | Ra
BIT_XOR	00011	Rd	Ra	00001	Rd <= Rd ^ Ra
BIT_AND	00011	Rd	Ra	00010	Rd <= Rd & Ra
BIT_NOT	00011	Rd	Ra	00011	Rd <= ~Ra
BIT_LSHFT	00011	Rd	Ra	00100	Rd <= Rd << Ra
BIT_RSHFT	00011	Rd	Ra	00101	Rd <= Rd >> Ra
MOV	00100	Rd	Ra	X	Rd <= Ra
MOVI	00101	Rd	i8		Rd <= i8
ARITH_U	**00110**	**Rd**	**Ra**	**s5**	**unsigned arithmetic**
ARITH_UADD	00110	Rd	Ra	11111	Rd <= uRd + uRa
ARITH_USUB	00110	Rd	Ra	10000	Rd <= uRd - uRa
ARITH_UADDI	00110	Rd	Ra	0AAAA	Rd <= uRd + Ra + AAAA
ARITH_S	**00111**	**Rd**	**Ra**	**s5**	**signed arithmetic**
ARITH_SADD	00111	Rd	Ra	11111	Rd <= sRd + sRa
ARITH_SSUB	00111	Rd	Ra	10000	Rd <= sRd - sRa
ARITH_SSUBI	00111	Rd	Ra	0AAAA	Rd <= sRd - sRa + AAAA
BR	**01000**	**Rd**	**i8**		**conditional branch**
BR_U	01000	Rd	0000 0000		Any
BR_E	01000	Rd	0000 0001		Z=1
BR_NE	01000	Rd	0000 0010		Z=0
BR_G	01000	Rd	0000 0011		Z=0 and S=O
BR_GE	01000	Rd	0000 0100		S=O
BR_L	01000	Rd	0000 0101		S != O
BR_LE	01000	Rd	0000 0110		Z=1 or (S != O)
BR_S	01000	Rd	0000 0111		S=1
BR_NS	01000	Rd	0000 1000		S=0
CMP	01001	Rd	Ra	X	SZO <= CMP(Rd, Ra)
SETC	01010	Rd	Imm8		Rd <= (Imm8 _f_ SZO) ? 1 : 0
MULT	01011	Rd	Ra	X	Rd <= uRd * uRa
HALT	01100	X			
LWEX	01101	Rd	Ra	s5	Rd <= RAM[Ra+s5]
SWEX	01110	Rd	Ra	s5	RAM[Ra+s5] <= Rd Rd <= 0|1 if success

SW Example

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

Main Project

B. Lancaster

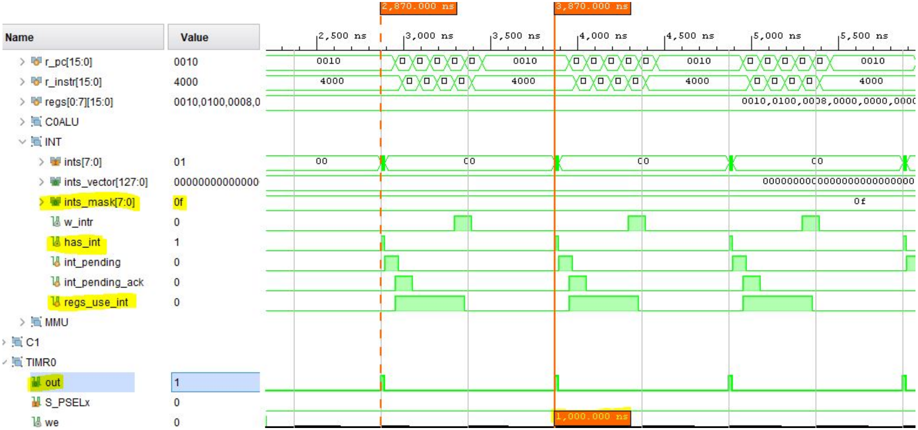
Introduction

Top Level Design
- Overview
- Memory Map
- Interconnect
- ISA
- Interrupts

Multi-core Functionality

Results

Conclusion



Name	Value
r_pc[15:0]	0010
r_instr[15:0]	4000
regs[0:7][15:0]	0010,0100,0008,0
C0ALU	
INT	
ints[7:0]	01
ints_vector[127:0]	00000000000000
ints_mask[7:0]	0f
w_intr	0
has_int	1
int_pending	0
int_pending_ack	0
regs_use_int	0
MMU	
C1	
TIMR0	
out	1
S_PSELx	0
we	0

Figure: TIMR0 1us interrupt with context switching

Timer Peripheral Registers

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Load Value																0200 RW
													I	R	S	0201 W
Prescaler																0202 W

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

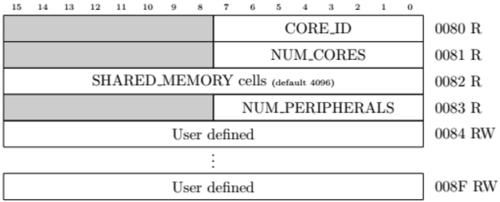
Hardware:

- **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



▨	CORE_ID	0080 R
▨	NUM_CORES	0081 R
SHARED_MEMORY cells (default 4096)		0082 R
▨	NUM_PERIPHERALS	0083 R
User defined		0084 RW
⋮		
User defined		008F RW

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
    nop
    nop

exit:
    halt
```

Atomic Instructions

- Enables semaphores, mutexes, memory barriers
- Prevent race conditions between threads/cores
- LW[EX] and SW[EX]
- Implementation in next slide

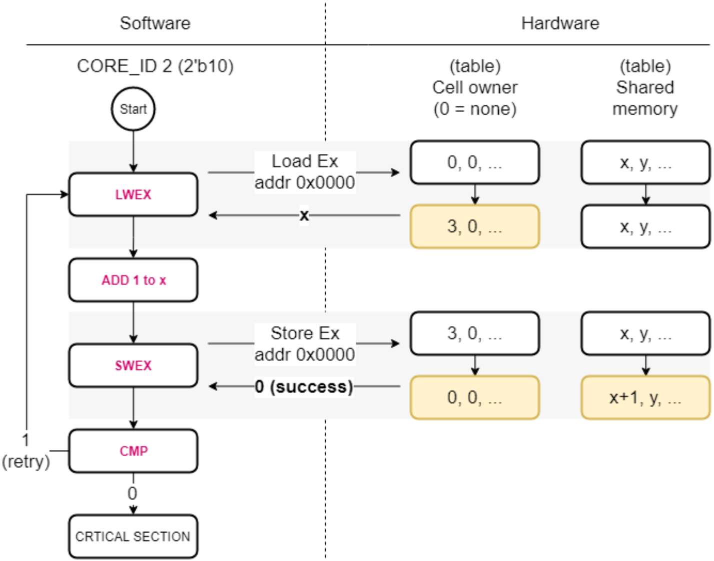
```
try_inc:
    // load and lock
    // (if not already locked)
    lwex    r0, r1
    // do something
    // (i.e. add 1 (semaphore))
    addi    r0, #0x01
    // attempt store
    swex    r0, r1

    // check success (== 0)
    cmp     r0, r3

    // if not equal (NE), retry
    movi    r4, try_inc
    br      r4, BR_NE

critical:
    // r0 is latest value
```

Exclusive Access Flow Chart



Software

Hardware

CORE_ID 2 (2'b10)

(table)
Cell owner
(0 = none)

(table)
Shared
memory

Start

LWEX

Load Ex
addr 0x0000

0, 0, ...

x, y, ...

x

3, 0, ...

x, y, ...

ADD 1 to x

SWEX

Store Ex
addr 0x0000

3, 0, ...

x, y, ...

0 (success)

0, 0, ...

x+1, y, ...

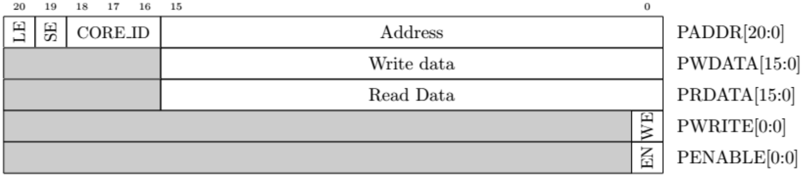
CMP

1
(retry)

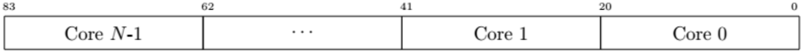
0

CRTICAL SECTION

HW - How do I know which core this lwex/swex is from?



The Core Idx is sent with each MMU request to the shared bus.



PADDR*NUMCORES-1:0 interconnect input.

Exclusive Access

```
mutex_claim:
    // load and lock
    // (if not already locked)
    lwex    r0, r1
    // do something
    // (i.e. add 1 (semaphore))
    addi    r0, #0x01
    // attempt store
    swex    r0, r1

    // check success (== 0)
    cmp     r0, r3

    // if not equal (NE), retry
    movi    r4, mutex_claim
    br      r4, BR_NE

critical:
    nop
```

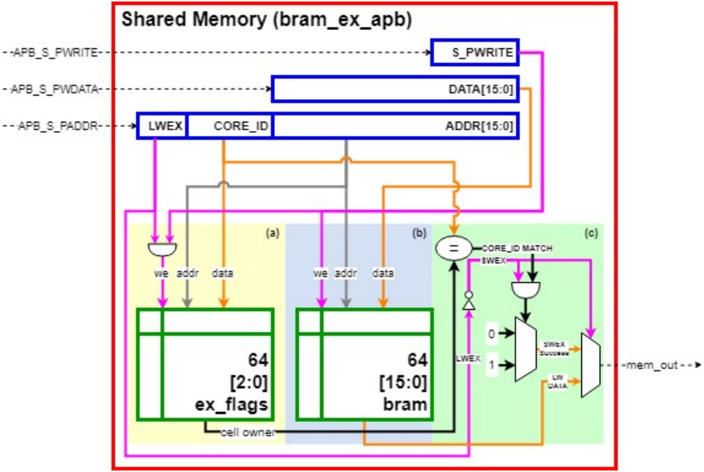


Figure: HW impl

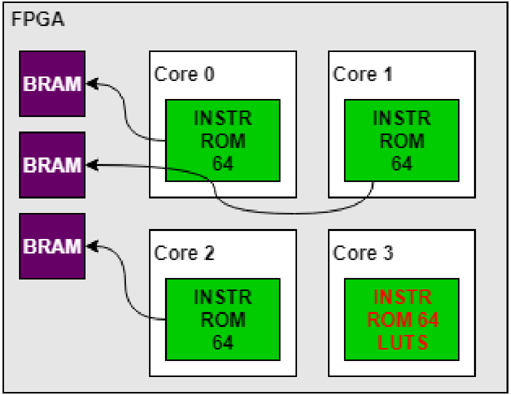
Demo: 8 core number summation (sum.s)

Design Challenges

Memory Limitations

Each core has it’s own instruction memory

- + Fast fetching and branching
- – Requires a dedicated BRAM (FPGA) per core
- – Limited BRAM blocks available
    - – Low consumption (`DEF_MEM_INSTR_DEPTH`)
    - – More cores = some implemented in LUT/regs (distributed)
- – **Reduces maximum core count**

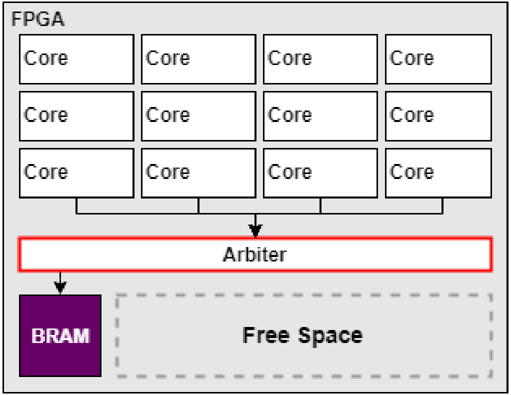


Design Challenges

Memory Limitations - Solution

Global instruction ROM

- + Reduce duplicate ROMs
- + Single BRAM requirement (expandable)
- + Smaller core size
- – **Slow access times** (exponential)
  - – Requires another interconnect/arbiter/scheduler = logic
- + **Increases maximum core count**



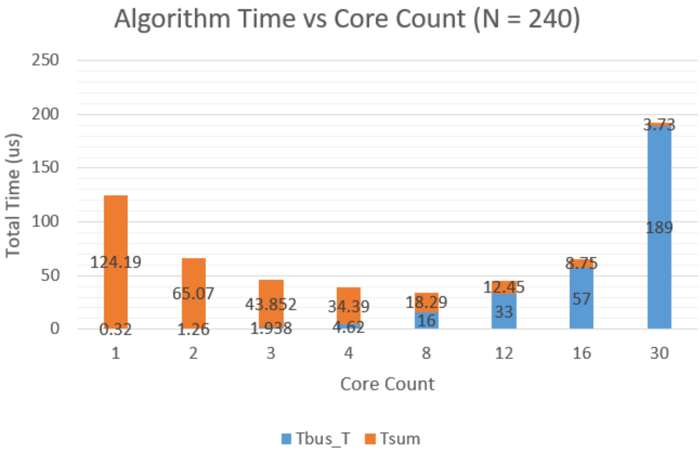
Summation - Multi-core vs Single-Core

- Each core has low work load
  - Sum subset of numbers in for loop

- Ideal scenario for parallelism
  - Highly parallelisable
  - Few inter-thread dependencies

Summation - Multi-core vs Single-Core

240 samples (@30 cores = 8 samples per core)



Main Project

B. Lancaster

Introduction

Top Level Design

Multi-core Functionality

Results

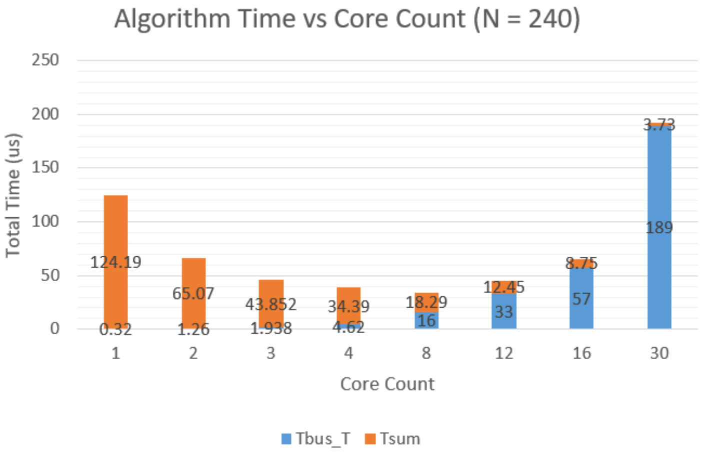
Summation

Per-core Memory VS Global Memory for instructions

Conclusion

Multi-core vs Single-Core for Summation

240 samples (@30 cores = 8 samples per core)



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

Info

- 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`
- Presentation tools:
  - Latex Beamer
  - `\usecolortheme{orchid}`
  - `\useoutertheme[hideothersubsections]{sidebar}`