Wide instep Instruction Set Machine Architecture

University of Cape Town

Oliver Funk (FNKOLI001)

Charles Schleich (SCHCHA027)

22/05/2017

EEE4084F

Abstract

A WIISMA processor chip is a multiple core architecture design that aims to execute wide instructions and implement the multiple instruction single data taxonomy in its operation.
The wide instructions of WIISMA come in the form of a single independent instruction for each processing core, all being read in and executed simultaneously on each of WIISMA's Acores.
The WIISMA architecture includes a single Control core (Ccore), and a set of Arithmetic cores (Acore), which run synchronously on each clock pulse, and was designed with Multiple instruction Single data operation in mind.

Contents

Introduction

This report covers the design decisions, and operation of a Wide Instep Instruction Set Machine Architecture Processor. This report includes a block diagram abstraction of the overall system, a draft schematic of a single Acore and a high level view of its internals, an in depth look into the instruction set for both the Acores and Ccores, as well as a programming model that will be implemented in the WIISMA system.
One of the major design decisions was to allow the Control Core to have access to the shared registers. This added a layer of complexity and increased the total number of bus connections from the shared register memory block, but was seen as necessary as this opened up many use cases for programs that required execution of programs to involve the C-Core in finalizing the execution cycle. This also allowed the Ccore to have control over storing the final result to memory.

The development process of this WIISMA was iterative much like the spiral model. The project started with critical analysis of a concept, if the concept was found to be feasible, it was taken forward for more indepth design. Once in the design and implementation phase, it was discussed if the implementation of the project would fit into the design requirement and if, considering design and resource restraints, it was feasible continue with this project.
The aim of this project was to conceptualize a WIISMA system, and so the implementation and testing phases were missed.
From there, the project was reviewed and a decision was made on whether or not it would be a worthwhile endeavor to continue with the development of the project.

One major discussion in each iteration of the design process was whether the benefit of adding a feature/core/processing unit/register to the overall system would outweigh the overall complexity that would come about with said feature being added to the system, i.e. complexity vs added value to the system as a whole, and in many cases this led to an idea being dismissed as the reward would be too insubstantial compared to the added complexity of said addition .

In developing WIISMA there were a number of associated risks, one such risk was ensuring that the design avoided hardware bottlenecks that would lead to unsatisfactory performance.
Another such risk is ensuring that the design criteria of the hardware is met and is feasible not only on paper but also in a physically implemented system.

WIISMA Processor Design

In the WIISMA system there are 4 Arithmetic Cores, each core operating independently from the others, on the same 32 bit word from memory. There is also a single Control core, which manages the operation of the chip and it's components.
There is a status register (SR) on the chip that is part of the 'shared registers' block abstraction and stores the status' of each of the most recent operation executed of each of the Acores.
The status register is 16 bits in length and dedicates 4 bits to represent the N,Z,C,V flags of each Acore. The Flags are explained further in the Instruction Set Planning section.

Arithmetic core Design

Each Acore was designed with 4 internal registers, to be used for intermediate calculation, and to store any constants that may be specific to the computation of that Acore. Each Acore has an adders, Multiplexers, Accumulators and an ALU for computation of the data block, MD. The Acores also have access to the shared registers SA, SB, SC, SD, as well as MP and MD, and their respective instruction register in the ICache. Upon completion of the

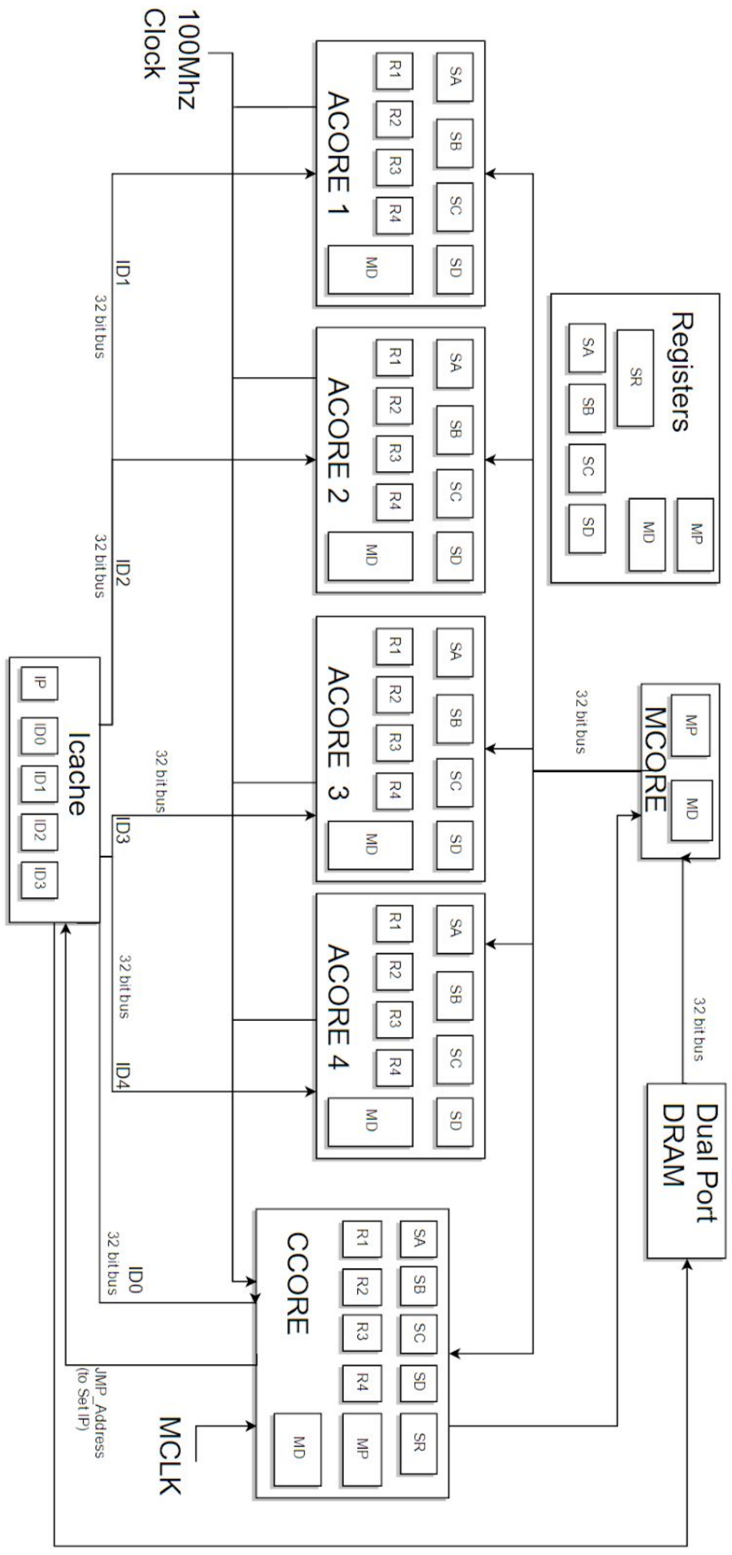
Control core Design

The Ccore has 4 internal 32 bit registers for storing intermediate solutions, used in the case of branching and branch prediction.
The Ccore has access to the shared registers SA, SB, SC, SD, as well as MP, this access is for retrieving the results of the computations from the Acores and storing them in memory if need be.

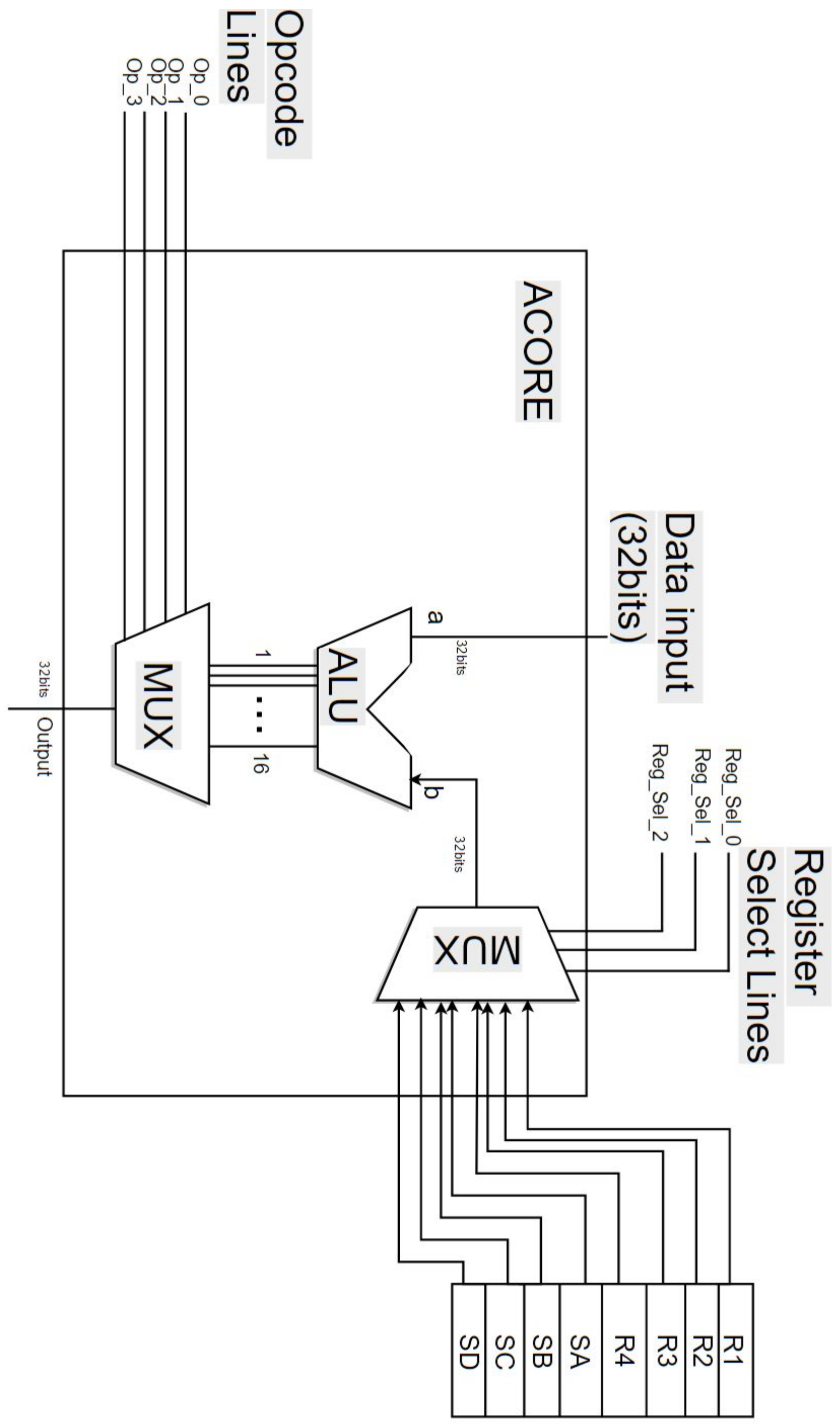
Bus Design

The width of all of the data-busses are 32 bits wide so that entire words can be moved between cores in a single clock pulse. This operation is parallel and not serial, in order to trade smaller bus width for a higher overall hardware throughput, specifically with respect to memory fetches. This means that one clock pulse will transfer the entire next data word from Mcore to each Acores, and that the Ccore can write the output word in one pulse. This also means that the Icache can update the instructions of each Acore simultaneously on one clock cycle.

Block Diagram of WIISMA



Schematic of Acore



WIISMA Instruction Set Planning

In the design of the WIISMA processor, a number of factors needed to be considered pertaining to how the cores will be programmed. As discussed previously, there will be four Arithmetic Cores (Acores) and one Control Core (Ccore). The Acores will only perform arithmetic operations and Ccore will be responsible for the storage, retrieval of data and loading of instructions. The instruction sets for the Acores and Ccore are distinct from each other, in that the same opcode can mean two different things on each core, however the assembler codes for the cores must be globally unique, they cannot share an assembler code.

Ccore Instructions

The following table shows the intructions for Ccore. These are the assembly to binary mappings, showing how the binary instruction is laid out.

Instruction	Description	Instruction **opcode**
NOP	No operation, core does nothing	0b**[00 01]**
STY	Suppress Mcore from incrementing the Memory Pointer (MP)	0b**[00 10]**
SW <**reg1**> <**reg2**>	Swap the the values in register **reg1** with the values in register **reg2**.	0b[reg2 register code] [reg1 register code] **[00 11]**
B <*label*>	Set the Instruction Pointer (IP) to the memory address of *label*.	0b[label memory address] **[01 00]**
B<**cond**> <**core**> <*label*>	Set the Instruction Pointer (IP) to the memory address of *label* only if the specified **condition** for a specific **core** is true. See the branching conditionals section below for the conditions.	0b[label memory address] [core code] [condition opcode] **[01 01]**
BR <*const*>	Set the Instruction Pointer (IP) to the memory address	0b[constant (28-bits)] **[01 10]**

	value of IP + *const*. *const* is a 28-bit number, where the MSB indicates the direction of the jump and the rest of the 27 bits are the value of the jump. This can be any number between $2^7 - 1$.	
BR<**cond**> <**core**> <*const*>	Set the Instruction Pointer (IP) to the memory address value of IP + *const*, only if the specified **condition** is true. *const* is a 29-bit number, where the MSB indicates the direction of the jump and the rest of the 28 bits are the value of the jump. This can be any number between $2^8 - 1$. See the branching conditionals section below for the conditions.	0b[constant] [core code] [condition opcode] **[01 11]**
LDR <**reg**> <*const*>	Load the value *const* into register **reg**, must be an internal register.	0b[reg register code] [**01 00**]
LDR <**reg1**> <**reg2>** <*offset*>	Load the value at the memory address stored in register **reg2** (must be an internal register), with an *offset* (positive only) into register **reg1**. This sets MP, MD and **reg1**.	0b[offset] [reg2 register code] [reg1 register code] [**01 01**]
STR <**reg1**> <**reg2**> <*offset*>	Store the value in register **reg1** into the memory address stored in register **reg2** (must be an internal register), with an *offset* (positive only). This sets MP and MD.	0b[offset] [reg2 register code] [reg1 register code] [**01 10**]

Acore Instructions

Instruction	Description	Instruction **opcode**
NOP	No operation, core does nothing	0b**[00 01]**
STY	Suppress Mcore from incrementing the Memory Pointer (MP)	0b**[00 10]**
MOV <**reg**> <*imm*>	Put the value *imm* into the **reg** register.	0b[imm value] [reg register code] **[01 00]**
MOV <**reg1**> <**reg2**>	Overwrite the value in the **reg1** register with the value in the **reg2** register.	0b[reg2 register code] [reg1 register code] **[01 01]**
CMP <**reg**> <*imm*>	Subtract the value *imm* from the value in the **reg** register (**reg** - *imm*). This updates the status flag for the core.	0b[imm value] [reg register code] **[01 10]**
CMP <**reg1**> <**reg2**>	Subtract the value in the **reg2** register from the value in the **reg1** register (**reg1** - **reg2**). This updates the status flag for the core.	0b[reg2 register code] [reg1 register code] **[01 11]**
ADD <**regR**> <**reg1**> <**reg2**>	Add the values in the **reg1** and **reg2** registers, put the result in **regR**. This updates the status flag for the core.	0b[reg2 register code] [reg1 register code] [regR register code] **[10 00]**
SUB <**regR**> <**reg1**> <**reg2**>	Subtract the value in the **reg2** register from the value in the **reg1** register (**reg1** - **reg2**), put the result in **regR**. This updates the status flag for the core.	0b[reg2 register code] [reg1 register code] [regR register code] **[10 01]**
MUL <**regR**> <**reg1**> <**reg2**>	Multiple the values in the **reg1** and **reg2** registers, put	0b[reg2 register code] [reg1 register code] [regR register code]

	the result in **regR**. This updates the status flag for the core.	**[10 10]**
NOT <**regR**> <**reg**>	Logical NOT the value in the **reg** registers, put the result in **regR**. This updates the status flag for the core.	0b[reg register code] **[11 00]**
AND <**regR**> <**reg1**> <**reg2**>	Logical AND the values in the **reg1** and **reg2** registers, put the result in **regR**. This updates the status flag for the core.	0b[reg2 register code] [reg1 register code] [regR register code] **[11 01]**
OR <**regR**> <**reg1**> <**reg2**>	Logical OR the values in the **reg1** and **reg2** registers, put the result in **regR**. This updates the status flag for the core.	0b[reg2 register code] [reg1 register code] [regR register code] **[11 10**
XOR <**regR**> <**reg1**> <**reg2**>	Logical XOR the values in the **reg1** and **reg2** registers, put the result in **regR**. This updates the status flag for the core.	0b[reg2 register code] [reg1 register code] [regR register code] **[11 11]**

All the Acore instructions take the same number of clock cycles to execute, in order to simplify coordination between cores.

Register Codes

The Acores and Ccore will each have four internal registers (R1 -> 4) and four external registers (RS -> D). These register are addressed using the following codes:

Core	Description	Core **opcode**
R1	Instructions relating to core **R1**	0b**[0 00]**
R2	Instructions relating to core **R2**	0b**[0 01]**
R3	Instructions relating to core **R3**	0b**[0 10]**
R4	Instructions relating to core **R4**	0b**[0 11]**

SA	Instructions relating to core **SA**	0b**[1 00]**
SB	Instructions relating to core **SB**	0b**[1 01]**
SC	Instructions relating to core **SC**	0b**[1 10]**
SD	Instructions relating to core **SD**	0b**[1 11]**

Core Codes

Some operations require a core to be specified (like branching), thus a way to specify which core the instruction relates to much be provided. All instructions that require a core as part of the opcode must use the following table.

Core	Description	Core **opcode**
A1	Instructions relating to core **A1**	0b**[00]**
A2	Instructions relating to core **A2**	0b**[01]**
A3	Instructions relating to core **A3**	0b**[10]**
A4	Instructions relating to core **A4**	0b**[11]**

Branching Conditionals

Internal status flags are set based on the result of certain operations. The values of the flags for each core are stored in the shared Status Register (SR). The first four bits of the SR are for core A1, the

- **N**: Set to 1 when the result of the operation is negative, otherwise cleared to 0
- **Z**: Set to 1 when the result of the operation is zero, otherwise cleared to 0
- **C**: Set to 1 when the operation results in a carry, otherwise cleared to 0.
- **V**: Set to 1 when the operation causes an overflow, otherwise cleared to 0.

The branching conditions are based on these flags in the following manner:

Condition	Flags	Description	Condition **opcode**
EQ	Z = 1	Equal. Last result was zero.	0b**[000]**
NE	Z = 0	Not equal.	0b**[001]**

		Last result was non-zero.	
GT	Z = 0 and N = V	Greater than (>). For signed numbers.	0b**[010]**
GE	N = V	Greater than or equal (≥). For signed numbers.	0b**[011]**
LT	N ! = V	Less than (<). For signed numbers.	0b**[100]**
LE	Z = 1 and N ! = V	Less than or equal (≤). For signed numbers.	0b**[101]**
PZ	N = 0	Positive or zero.	0b**[110]**
NE	N = 1	Negative.	0b**[111]**

*These are based on the condition flags for the application program status register (APSR) in the STM32F0.

WIISMA Programming

If any of the Acores experience a hard fault or exception, the Ccore branches to the memory address 0x00000000 to handle exceptions.

The decision was made to allow the Ccore access to all of the shared registers SA, SB, SC, SD, this was done to allow the Ccore to access the final result of the computation, so that it may write said result to a memory location.

Code example

An MISD architecture could be well suited to a checksum algorithm. The single data input could be a security code, credit card number or something similar. Each core could independently manipulate the number, each performing one part of the checksum algorithm. Finally, the manipulations can get recombined to produce a result.
In the following example, some 32-bit number is maintained by each core. Acore1 subtracts 1 from the number, Acore2 adds 1 to the number, Acore3 multiples the number by 2 and Acore3 performs a logical NOT on the number. Each of these results are stored in the shared registers. A recombination process then occurs, the results from Acore1 and Acore2 undergo an XOR operation together by Acore1 and the results

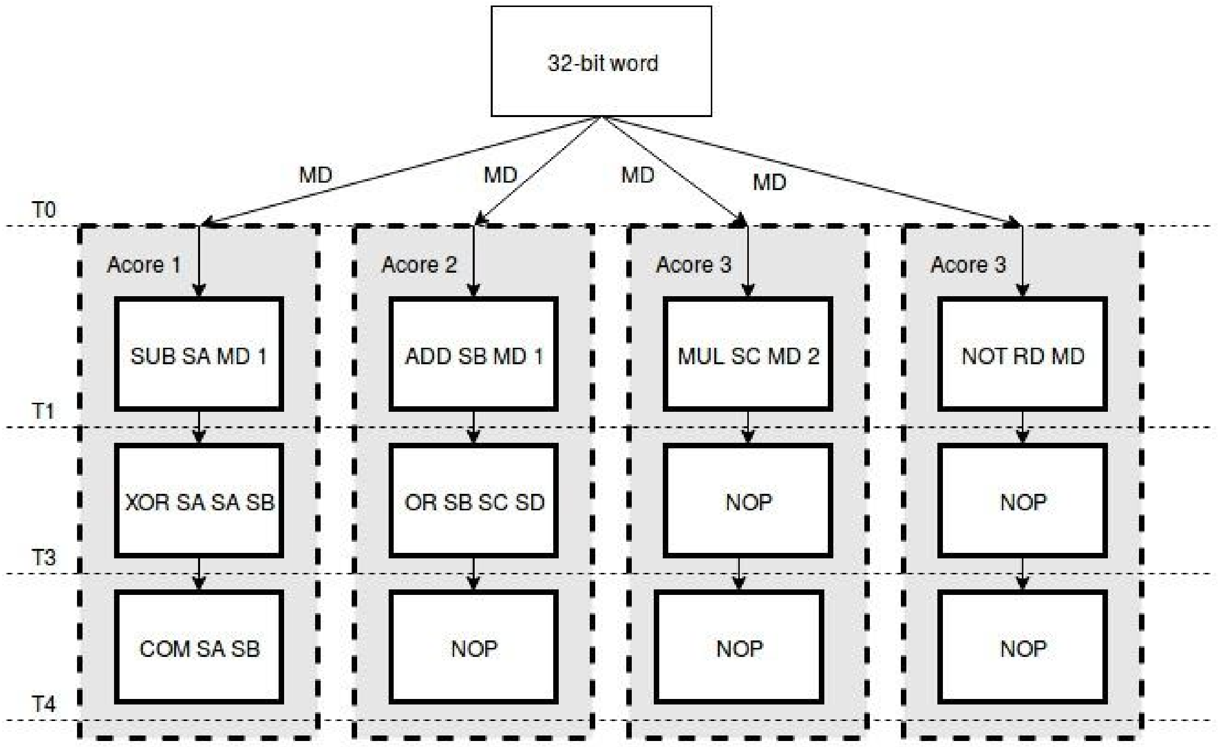
from Acore3 and Acore4 undergo an AND operation together by Acore2. Following that the two results are compared, checking if the one is bigger than the other, which sets a status flag. The Ccore will then check this status flag and write a value to memory, indicating whether or not the original number was valid.

C-code example

```
public void checksum(){
        SA = numbers[i] - 1;
        SB = numbers[i] + 1;
        SC = numbers[i] * 2;
        SD = ~(numbers[i]);

        If ( (SA^SB) > (SC | SD) ){
                is_valid = true;
        else{
                is_valid = false;
        }
}
```

The operation of the Acores for the main part of the algorithm is illustrated in the flow diagram below.



Assembly code example

*assuming the memory address **numbers** points is stored in R1, the memory address of **is_valid** is stored in R3 and i is some number stored in R4 *

```
:return
LDR R2 R1 1            #[on Ccore] updates MP to numbers memory address and MD to
numbers[i]
SUB SA MD 1            #[on Acore 1]
ADD SB MD 1            #[on Acore 2]
MUL SC MD 2            #[on Acore 3]
NOT SD MD              #[on Acore 4]
```

#IP is incremented by the number of Acore + number of Ccore multiplied by 4 because each register in the instruction memory is a byte long and the instructions are 32-bit long, therefore 4 bytes are needed for one instruction -> 4 + 1 = 5*4=20
#IP is incriemnted internally by hardware

```
STY                    #[on Ccore]
XOR SA SA SB           #[on Acore 1]
OR SB SC SD            #[on Acore 2]
NOP                    #[on Acore 3]
NOP                    #[on Acore 4]
```

#IP is incremented

```
STY                    #[on Ccore]
CMP SA SB              #[on Acore 1]
NOP                    #[on Acore 2]
NOP                    #[on Acore 3]
NOP                    #[on Acore 4]
```

#IP is incremented

```
BGT A1 is_greater      #[on Ccore] the status flag for Acore1 is checked for a greater than
comparison, which if taken changes IP
NOP                    #[on Acore 1]
NOP                    #[on Acore 2]
NOP                    #[on Acore 3]
STY                    #[on Acore 4]
```

#IP is incremented

```
BLT A1 is_lessthan     #[on Ccore] the status flag for Acore1 is checked for a greater than
comparison
NOP                    #[on Acore 1]
NOP                     #[on Acore 2]
NOP                     #[on Acore 3]
STY                     #[on Acore 4]

:is_greater
LDR R2 1                #[on Ccore]
NOP                     #[on Acore 1]
NOP                     #[on Acore 2]
NOP                     #[on Acore 3]
STY                     #[on Acore 4]

STR R2 R3 0             #[on Ccore]
NOP                     #[on Acore 1]
NOP                     #[on Acore 2]
NOP                     #[on Acore 3]
STY                     #[on Acore 4]

B return                #[on Ccore]
NOP                     #[on Acore 1]
NOP                     #[on Acore 2]
NOP                     #[on Acore 3]
NOP                     #[on Acore 4]

#By not calling STY, the MP is incirmented automatically by hardware

:is_less
LDR R2 0                #[on Ccore]
NOP                     #[on Acore 1]
NOP                     #[on Acore 2]
NOP                     #[on Acore 3]
STY                     #[on Acore 4]

STR R2 R3               #[on Ccore]
NOP                     #[on Acore 1]
NOP                     #[on Acore 2]
NOP                     #[on Acore 3]
STY                     #[on Acore 4]

B return                #[on Ccore]
NOP                     #[on Acore 1]
NOP                     #[on Acore 2]
NOP                     #[on Acore 3]
NOP                     #[on Acore 4]

#By not calling STY, the MP is incirmented automatically by hardware
```

Discussion and Conclusion

The solution designed was the result of multiple iterations of planning and discussion, and resulted in a fairly well optimised chip from a hardware point of view. The fact that the busses would move data all in parallel would be a benefit for speed of moving data out of memory and between registers especially in expensive memory fetch operations, but would also lead to a more complex circuit pathing design as well as a higher burst of EMI on every read or write operation. The other approach, being serial communication, would result in a much simpler circuit path layout, and lower burst EMI, with the tradeoff of memory fetches and communication between cores being very slow, and lowering overall utilization of the chip.
Another area which the design could be improved upon would be the status register (SR), where 16 of the bits go unused. This could be dealt with by implementing 8 Acores instead of 4, such that the 8 Acores would each require 4 flags, and so 32 bits would be needed to properly represent the status of operations of each of the cores. One final area in which the design could be improved, is that there are 13 opcodes in the Acore instruction set. 4 is the smallest number of bits used to represent 13 instructions, however 4 bits can represent 16 instructions which means there is space for 3 more instructions. To better utilize the opcode bits, 3 more possibly slightly more complex instructions could be added to the Acores instruction set.

The instruction set of the WIISMA processor was designed to make efficient use of the cores operations, while the opcodes were designed to make efficient use of the number of bits in representation of said instructions. The Acore and Ccore have independant instruction sets as their functionality is catered towards different operational needs of the system. The flags used in WIISMA are based off of the APSR flags used in the STM32F0.

A system such as this would not be used widely in industry for a number of reasons.
The whole paradigm of a multiple instruction single data system does not apply well to real world applications. Very specialized applications may make use of a MISD taxonomy, such as applications that require high fault tolerance, or applications that my need to test a certain block of data, in different ways that are expected to arrive at the same conclusion.

References

1. Oak Ridge National Laboratory, 3.1 Flynns Taxonomy [by Nikos Drakos Aug, 1994] Available from: https://www.phy.ornl.gov/csep/ca/node11.html [Accessed 21 May 2017]
2. Lecture 10. (2017). [slideshow] Cape Town: Simon Winberg, pp.10 - 25. Available at: http://www.rrsg.ee.uct.ac.za/courses/EEE4084F/Resources/Slides/EEE4084F-Lecture10.pptx [Accessed 21 May 2017].