

Fusion-Core ISA Definition: Revision 0.1

Dylan Wadler

October 23, 2017

Contents

1	Change log	4
2	Introduction	4
2.1	About	4
2.2	Goals	4
2.3	Conventions	4
3	Register File Definitions	4
3.1	Register File List	5
3.2	General Purpose Registers	5
3.3	Special Registers	5
3.3.1	Control Registers	5
3.3.2	Supervisor Registers	6
3.4	Adding Registers	6
4	Instruction Definitions	6
4.1	Instruction Types	6
4.1.1	Integer	6
4.1.2	Immediate	6
4.1.3	Load/Store	6
4.1.4	Branch/Jump	6
4.1.5	Co-Processor	6
4.2	List of Instructions	6
4.2.1	Integer	6
4.2.2	Immediate	6
4.2.3	Load/Store	6
4.2.4	Branch/Jump	6
4.2.5	Co-Processor	6
5	Exceptions and Interrupts	6
5.1	Exceptions	6
5.2	Interrupts	6
5.2.1	User Level	6
5.2.2	Supervisor Level	6
6	Co-Processors	6
6.1	Co-Processor Interface	7
6.1.1	Co-Processor Conventions	8
6.1.2	Register Connections	8
6.1.3	Decode unit Connections	8
6.2	Interface Connection Definitions	8
6.3	Adding custom Co-Processor	8
6.4	List of Co-Processors	8
6.4.1	Floating Point	8
6.4.2	System Unit	8
6.4.3	Memory Management Unit	8
6.4.4	Multiprocessor Commnuication Unit	8

7	Recommended Co-Processors	8
7.1	Floating Point Unit	8
7.1.1	Registers	8
7.1.2	Instructions	8
7.2	System Unit	8
7.2.1	Registers	8
7.2.2	Instructions	8
7.3	Memory Management Unit	8
7.3.1	Registers	8
7.3.2	Instructions	8
7.4	Multiprocessor Communication Unit	8
7.4.1	Registers	8
7.4.2	Instructions	8
8	Memory Map	8
9	Programming Conventions	9
9.1	Register Usage	9
9.2	Memory Locations for Vector Table	9
9.2.1	Interrupt Vector Table	9
9.2.2	Exception Vector Table	9

1 Change log

Version 0.1 Initial Definition of the Instruction Set Architecture

2 Introduction

2.1 About

The Fusion-Core ISA is dedicated to creating an easily expandible architecture without altering the instruction set. By use of defining an easy interface with a simple core instruction set, this allows for more freedom in implementation. High end processors and microcontrollers would only have slight variations in configuration, as their core would remain identical save for easy to maintain and scalable co-processors.

The architecture is Big endian, with a core instruction set that is RISC, but the co-processors do not need to adhere to the RISC philosophy. This allows for more flexibility in design, and possibly faster core clock speeds as the pipeline would depend on smaller amounts of logic. Only the instructions provided in this document are to be implemented in the main processor. The co-processors defined in this document are recommended, but not required for normal function. Co-processor documentation is to be provided by the creator, and should adhere to the standards of clarity and conciseness such that it can be easily implemented from the documentation alone in a HDL.

64 Bit instructions: At this moment in time, the Fusion-Core ISA is only a 32 bit ISA. Due to the focus on co-processors, older implementations could easily be modified to include 64 bit operations.

2.2 Goals

2.3 Conventions

Document Conventions: Example code will be shown with `monospace` text. General purpose registers will be denoted with `$R#` where `#` is the number of the register. Special purpose registers will be presented with **bold** text.

Naming Conventions: The name of input signals will have `_in` after the signal name, with `_out` after output signals. This is mainly used in the verilog example implementation. If a naming convention is not globally used, it will be stated in the individual section that it pertains to.

3 Register File Definitions

This section goes over the different registers available in the ISA. Each register file name begins with `"REGF"`, such as the first General Purpose Register File being `REGFGP0`. Any additional register files require the number after the name of the register file. Register files with additional numbers after them are bank switched to reduce space, hence why the number is required to denote the register file space used.

To alleviate context switching delays, three general purpose register files are bank available by bank switching. Only these register files are required for a minimal system, though more can certainly be implemented if required. Each register file can be accessed by changing the value in the BSELRF register.

3.1 Register File List

Figure 1: General Purpose Registers

REGFGP0		REGFEXCP		REGFSYSCLO	
Register	Register Name	Register	Register Name	Register	Register Name
\$R0	ZERO	\$R0	ZERO	\$R0	ZERO
\$R1	SP0	\$R1	SP1	\$R1	SP1
\$R2	FP0	\$R2	FP1	\$R2	FP1
\$R3	GP0	\$R3	GP1	\$R3	GP1
\$R4	RA0	\$R4	RA1	\$R4	RA1
\$R5	ARG00	\$R5	ARG0	\$R5	SYSARG00
\$R6	ARG01	\$R6	ARG1	\$R6	SYSARG01
\$R7	ARG02	\$R7	ARG2	\$R7	SYSARG02
\$R8	ARG03	\$R8	ARG3	\$R8	SYSARG03
\$R9	ARG04	\$R9	ARG4	\$R9	SYSARG04
\$R10	ARG05	\$R10	ARG5	\$R10	SYSARG05
\$R11	RVAL00	\$R11	RVAL0	\$R11	SYSRVAL00
\$R12	RVAL01	\$R12	RVAL1	\$R12	SYSRVAL01
\$R13	RVAL02	\$R13	RVAL2	\$R13	SYSRVAL02
\$R14	RVAL03	\$R14	RVAL3	\$R14	SYSRVAL03
\$R15	RVAL04	\$R15	RVAL4	\$R15	SYSRVAL04
\$R16	GPR00	\$R16	GPR0	\$R16	SYSGPR00
\$R17	GPR01	\$R17	GPR1	\$R17	SYSGPR01
\$R18	GPR02	\$R18	GPR2	\$R18	SYSGPR02
\$R19	GPR03	\$R19	GPR3	\$R19	SYSGPR03
\$R20	GPR04	\$R20	GPR4	\$R20	SYSGPR04
\$R21	GPR05	\$R21	GPR5	\$R21	SYSGPR05
\$R22	GPR06	\$R22	GPR6	\$R22	SYSGPR06
\$R23	GPR07	\$R23	GPR7	\$R23	SYSGPR07
\$R24	TMPR00	\$R24	TMPR0	\$R24	SYSTMPR00
\$R25	TMPR01	\$R25	TMPR1	\$R25	SYSTMPR01
\$R26	TMPR02	\$R26	TMPR2	\$R26	SYSTMPR02
\$R27	TMPR03	\$R27	TMPR3	\$R27	SYSTMPR03
\$R28	TMPR04	\$R28	TMPR4	\$R28	SYSTMPR04
\$R29	TMPR05	\$R29	TMPR5	\$R29	SYSTMPR05
\$R30	REGHI0	\$R30	REGHI1	\$R30	SYSREGHI0
\$R31	REGLOW0	\$R31	REGLOW1	\$R31	SYSREGLOW0

3.2 General Purpose Registers

32 general purpose registers that are 32 bits wide are available, as shown in Figure 1, in the previous section.

NOTE: In writing programs, register arguments and return values can be used to reduce the number of registers used between processes.

While it is not defined by the architecture, larger general purpose registers can be used instead of 32 bit wide registers. If larger registers are needed, consider using a co-processor to for instructions that require larger operands. This provides code compatibility between different implementations.

3.3 Special Registers

3.3.1 Control Registers

Bank Switching Bank switching is controlled by the **BSELRF** register

3.3.2 Supervisor Registers

3.4 Adding Registers

4 Instruction Definitions

4.1 Instruction Types

4.1.1 Integer

4.1.2 Immediate

4.1.3 Load/Store

4.1.4 Branch/Jump

4.1.5 Co-Processor

4.2 List of Instructions

4.2.1 Integer

4.2.2 Immediate

4.2.3 Load/Store

4.2.4 Branch/Jump

4.2.5 Co-Processor

5 Exceptions and Interrupts

5.1 Exceptions

5.2 Interrupts

5.2.1 User Level

5.2.2 Supervisor Level

6 Co-Processors

Co-processors are the main point of the Fusion-Core architecture. As the ISA only defines the main core, the implementator is free to use whichever co-processors that would be necessary for an application. Hardware acceleration for vector instructions, encryption, floating point, communication, etc. There is no limitation for what kind of co-processor that could be used, only the number that could fit within the defined usable instructions.

It is important to note that the co-processors do not need to be of a RISC construction, due to this reliance on co-processors without specifying how they should be implemented. The main core is indeed RISC, as only the simplest instructions are defined that something as small as a microcontroller could use without issue.

The idea behind the separation of processor is to create different pipelines for each core. In doing so, the main core could be clocked faster than that of a pipeline requiring integer multiplication, or some other time consuming operation. As well, the individual cores could be clocked at their respective fastest frequencies, thus resulting in the fastest possible performance of each part. To deal with writing to memory with varying clock speeds for each core, a FIFO buffer is used to send each write to main memory. This FIFO is to create the illusion of write atomicity. The FIFO should have connections to allow for a seeming atomic read though, the programmer should note that reads require care; if there is a dependency on a slow core's written value. This programming paradigm is similar to that of parallel threads, where certain values may not be available until an unknown time. The ISA does not define any particular ways to handle this and it is either left up to the implementation, or programmer depending.

As shown in the instruction section, there are predefined regions for the co-processor slots. This is defined to allow for the compiler/assembler to work across implementations. At this time, only a fixed number of co-processors can be used on an implementation at a time, though future expansions to the ISA may change this.

6.1 Co-Processor Interface

The interface is designed to be extremely simple, as to make co-processors easy to implement. The inputs is just the output from the instruction fetch passthrough on the decode unit. As shown in the instruction list, there is a section of 6 bits for the co-processor's opcode. Please also note that this section of the instruction is different than the opcode, 0x3f (may change to 0x20 to allow for more opcodes for the co-processors), which indicates that the instruction is for the co-processor. To save connections, the normal opcode section is removed, leaving 26 bit physical instructions. Using more bits per instruction is not supported at this time, though it is possible to take the instruction fetch output directly.

- 6.1.1 Co-Processor Conventions
- 6.1.2 Register Connections
- 6.1.3 Decode unit Connections
- 6.2 Interface Connection Definitions
- 6.3 Adding custom Co-Processor
- 6.4 List of Co-Processors
 - 6.4.1 Floating Point
 - 6.4.2 System Unit
 - 6.4.3 Memory Management Unit
 - 6.4.4 Multiprocessor Communication Unit

7 Recommended Co-Processors

- 7.1 Floating Point Unit
 - 7.1.1 Registers
 - 7.1.2 Instructions
- 7.2 System Unit
 - 7.2.1 Registers
 - 7.2.2 Instructions
- 7.3 Memory Management Unit
 - 7.3.1 Registers
 - 7.3.2 Instructions
- 7.4 Multiprocessor Communication Unit
 - 7.4.1 Registers
 - 7.4.2 Instructions

8 Memory Map

NOTE This section may not remain and be left up to the implementation. Do not use without consulting the documentation of the implementation.

9 Programming Conventions

9.1 Register Usage

9.2 Memory Locations for Vector Table

9.2.1 Interrupt Vector Table

9.2.2 Exception Vector Table

Address (32 bit)	Definition
0x0000	Reset address
0x0000	
0x0000	
0x0000	
0x0000	
0x0000	

Figure 2: Exception Vector Table