Fusion-Core ISA Definition: Revision 0.1

Dylan Wadler

December 24, 2017

Contents

1 Change log

Version 0.1 Initial Definition of the Instruction Set Architecture

2 Introduction

2.1 About

Introduction The Fusion-Core ISA is dedicated to creating an easily expandible architecture without having to recompile a program's binary. 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 require slight varaitions in configuration, as their core would remain identical save for easy to maintain and scalable co-processors.

Main Ideas 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.

Co-Processors The Co-Processor interface is currently defined by setting the MSB within the OP Code field of an instruction, to decrease complexity of the Decode unit. In doing so, this allows for coprocessor code to be written in the same memory space as the main processor code. In the current iteration, up to 32 different coprocessors can be used, with the option for dynamic or static allocation of the OP Codes. The interface for the coprocessors is explained further in the dedicated section.

2.2 Goals

The main goal is to provide an architecture with a simple decoding unit and the ability to utilize a single binary for all implementations of the architecture.

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 written with **bold** text.

Part I
Programming Information

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 availble 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	
Register	Register Name
$R0	ZERO
$R1	SP0
$R2	FP0
$R3	GP0
$R4	RA0
$R5	ARG0
$R6	ARG1
$R7	ARG2
$R8	ARG3
$R9	RVAL0
$R10	RVAL1
$R11	GR0
$R12	GR1
$R13	GR2
$R14	GR3
$R15	GR4
$R16	GR5
$R17	GR6
$R18	GR7
$R19	GR8
$R20	GR9
$R21	GR10
$R22	TMP0
$R23	TMP1
$R24	TMP2
$R25	TMP3
$R26	TMP4
$R27	TMP5
$R28	TMP6
$R29	TMP7
$R30	HI0
$R31	LOW0

REGFSYSCL0	
Register	Register Name
$R0	ZERO
$R1	SP1
$R2	FP1
$R3	GP1
$R4	RA1
$R5	SYSARG0
$R6	SYSARG1
$R7	SYSARG2
$R8	SYSARG3
$R9	SYSARG4
$R10	SYSARG5
$R11	SYSRVAL0
$R12	SYSRVAL1
$R13	SYSRVAL2
$R14	SYSRVAL3
$R15	SYSRVAL4
$R16	GPR0
$R17	GPR1
$R18	GPR2
$R19	GPR3
$R20	GPR4
$R21	GPR5
$R22	GPR6
$R23	GPR7
$R24	SYSTMPR0
$R25	SYSTMPR1
$R26	SYSTMPR2
$R27	SYSTMPR3
$R28	SYSTMPR4
$R29	SYSTMPR5
$R30	SYSREGHI0
$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. There is a distinction between the System Register File and the General Purpose

Figure 2: Special Registers

System Registers			
Register Name	Description	Width (bytes)	Address (Hex)
CPUREV	CPU Revision	1	0x00000000
CPNUM	Co-Processor Number	1	0x00000001
CPO0	Co-Processor 0 ID	2	0x00000002
CPO1	Co-Processor 1 ID	2	0x00000004
CPO2	Co-Processor 2 ID	2	0x00000006
CPO3	Co-Processor 3 ID	2	0x00000008
STAT	Status Register	1	0x0000000a
n/a	RESERVED	1	0x0000000b
OPCAR	Opcode Allocation Pointer	4	0x0000000c

Figure 3: Supervisor Registers

Supervisor Registers		
Register Name	Description	Address (Hex)
SMSTAT	Supervisor mode status register	0x00000000
PRCPR0	Process Pointer register 0	0x00000004

Register File, as during certain syscall instructions, the register files are bank switched. Only GP0 through GP7 are saved, for passing between the different banks.

While it is not defined by the architecture, larger general purpose registers can be used instead of 32 bit wide registers. The System Register File allows 8 bit addressing for the registers to be accessed, in order to utilize more of the memory space. 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

The special registers are sorted between the System Registers and the Supervisor Registers. The system registers provide simple configuration values and some read-only registers to give the programmer information about the implementation. The registers defined in this manual are the bare minimum special purpose registers, and should be included for code compatibility. The supervisor registers

3.3.1 System Registers

STAT: Status Register

Z	OV	SPV	PEM		

STAT Read only register for various processor state information. The flags are explained in detail below.

Z

OV

PML

PEM

3.3.2 Supervisor Registers

3.4 Adding Registers

4 Memory

At this point in time, the ISA only handles 32 bit addresses. With memory capacity increasing in size as time goes on, this may change. In order to access larger sections of memory, the use of an offset register may be implemented to create 64 bit addresses. The use of virtual memory with said offset register is also accepted.

4.1 Memory Locations for Vector Table

4.1.1 Interrupt Vector Table

4.1.2 Exception Vector Table

Address (32 bit)	Definition
0x0000	Reset
0x0004	Co-Processor Microcode Exception
0x0008	
0x000c	
0x0010	
0x0014	

Figure 4: Exception Vector Table

Part II
Instructions

5 Instruction Definitions

5.1 Instruction Format Types

This section will talk about the different instructions available in the core processor, their encodings, function, and hazards they cause or registers they affect in the processor. The Co-Processor instructions are purlely generic and allow the implementation of each Co-Processor to determine

how their respective instructions will be decoded. Only the Operation Codes will be defined for each Co-Processor slot to allow for more customization.

5.1.1 Integer

The interger instructions are the heart of this processor's arithmetic abilities and is vital to ensure fast execution. A semi-strict adherence to RISC philosophy in this architecture is required to exploit any benefits to this ISA in a real implementation.

The integer instruction coding with descriptions, is shown in the diagrams below.

Register/Integer Instruction Format

opcode (0x01)	rd	rsa	rsb	shft	aluop
31 26	25 21	20 16	15 11	10 4	3 0

Overview: The Register/Integer Instructions are basic ALU operations, without immediates. Registers RSa and RSb are the two operands,which are stored in register Rd. The 4 bit ALUOP field denotes the settings for the ALU, to reduce complexity of selecting what operation to choose. The shft bits are only for the shift amount with the shifting instructions, but unused for other instructions.

The following instructions use this encoding:

5.1.2 Immediate

Immediate Instruction Format

opcode	rd	rsa	Immediate	aluop
31 26	25 21	20 16	15 4	3 0

5.1.3 Load/Store

Load Instruction Format

opcode	rd	rsa	Immediate
31 26	25 21	20 16	15 0

Load Immediate Instrution Format

opcode	rd	DSEL	Immediate
31 26	25 21	20 16	15 0

Store Instruction Format

opcode	Immediate[15:11]	rsa	rsb	Immediate[10:0]
31 26	25 21	20 16	15 11	10 0

5.1.4 Branch/Jump

Jump Instruction Format

opcode	rd	rsa	Immediate
31 26	25 21	20 16	15 0

Branch Instruction Format

opcode	Immediate[11:7]	rsa	rsb	Immediate[6:0]	aluop
31 26	25 21	20 16	15 11	10 4	3 0

System Instruction Format

opcode	rd	rsa	Function	Immediate
31 26	25 21	20 16	15 8	7 0

5.1.5 Co-Processor

Since the coprocessors can have any implementation, only the opcode is required. However, adhering to similar formatting of the instruction formats provided in previous sections is imperative and the Fusion-Core foundation will not provide an accepted coprocessor ID number. More information about coprocessors can be accessed in the Co-Processor section.

5.2 List of OPCodes

5.3 List of Instructions

5.3.1 Integer

5.3.2 Immediate

5.3.3 Load/Store

5.3.4 Branch/Jump

5.3.5 Co-Processor

6 Exceptions and Interrupts

6.1 Exceptions

6.2 Interrupts

6.2.1 User Level

6.2.2 Supervisor Level

Part III
Co-Processors

7 Co-Processors

Co-processors are the main point of the Fusion-Core archetecture. 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 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 dependancy 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.

7.1 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.

7.1.1 Co-Processor Conventions

7.1.2 Register Connections

7.1.3 Decode unit Connections

7.2 Interface Connection Definitions

7.3 Adding custom Co-Processor

7.4 List of Co-Processors

7.4.1 Floating Point

7.4.2 System Unit

7.4.3 Memory Management Unit

7.4.4 Multiprocessor Communication Unit

8 Recommended Co-Processors

About This section will cover some basic coprocessors that have been approved and assigned coprocessor IDs. The full list of approved coprocessors will be included in a separate document.

8.1 Floating Point Unit

8.1.1 Registers

8.1.2 Instructions

8.2 System Unit

8.2.1 Registers

8.2.2 Instructions

8.3 Memory Management Unit

8.3.1 Registers

8.3.2 Instructions

8.4 Multiprocessor Communication Unit

8.4.1 Registers

8.4.2 Instructions