Fusion-Core ISA Definition: Revision 0.2.1

Dylan Wadler

July 2, 2018

Contents

Part I

Administrative

1 Change log

Version 0.2.1 Updated Load Immediate DSEL values to create simpler decode control signals. Fixed various issues in document to be consistent with other sections. The binary representation table, Load Immediate, and System sections were modified.

Version 0.2 Added comprehensive list of Instruction usage for programmers. Changed opcode definitions to create simpler decode, with availability to create new instructions from the existing opcodes. Including more information about special purpose registers.

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 expandable 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 variations 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 co-processor code to be written in the same memory space as the main processor code. In the current iteration, up to 32 different co-processors can be used, with the option for dynamic or static allocation of the OP Codes. The interface for co-processors 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. In order to do this, a strict implementation of the main core is outlined in this document, along with registering co-processors to provide consistency. Mechanisms to allow for co-processor instructions to be executed in software are also available; more information is available in the respected section.

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 II

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.

3.1 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 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.

Figure 1: General Purpose Registers

GPREGF	
Register	Register Name
$R0	ZERO
$R1	SP
$R2	FP
$R3	GP
$R4	RA
$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

SYSREGF	
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 Special Purpose 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 are aimed at higher level functions required for operating system environments. They are not essential for operation, and can either be partially implemented or not at all. The optional parts will be noted in the register descriptions.

Figure 2: System Registers

System Registers			
Register Name	Description	Width (bytes)	Address (Hex)
CPUREV	CPU Revision	1	0x00000000
CPNUM	Co-Processor Number	1	0x00000001
STAT	Status Register	1	0x00000002
n/a	RESERVED	1	0x00000003
OPCARP	Opcode Registration Table Pointer	4	0x00000004
CPIDTP	Co-Processor ID Table Pointer	4	0x00000008
UINTEN	User Interrupt Enable	4	0x0000000c
RPTINFO	Running Process Thread Info	4	0x00000010
RPID	Running Process ID	2	0x00000012

Figure 3: Supervisor Registers

Supervisor Registers		
Register Name	Description	Address (Hex)
SMSTAT	Supervisor mode status	0x00000000
RPCPR0	Running Process Pointer	0x00000004
RPITP	Running Process Info Table Pointer	0x00000008
HTINFOP	Hardware Thread Info Table Pointer	0x0000000c
HTCTLP	Hardware Thread Control Table Pointer	0x00000010
ECODE	Exception Code	0x00000014
SYSCTP	System Call Table Pointer	0x00000015
n/a	RESERVED	0x00000018

3.3 Special Purpose Register Definitions

3.3.1 System Registers

CPUREV: CPU Revision Register

7 4	3 0
Major CPU Revision	Minor CPU Revision

The **CPUREV** register holds the revision number for the processor implementation. The upper 4 bits hold the major revision number, the lower 4 bits the minor revision number. This number is hard coded by the implementation. The major revision number should refer to the company implementation, with the minor revision number is for the implementation's revision.

Any permission level can read this register. Since it should be hard coded, there is no way to write to this register.

CPNUM: Co-Processor Number Register

7 0
Number of Co-processors

The **CPNUM** register simply holds the number of different types of co-processors in the processor implementation. This register is used for finding out how many CPIDs to check, for determining whether to use microcode or the co-processor hardware available for each co-processor.

This register can be read by with any permission level. This register should be hard coded, so no writes are possible.

STAT: Status Register

7	6	5	4	3	2	1	0
Z	OV	PEMA	PML2	PML1	PML0	INTN	SPCP

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

Z Zero Flag; Indicates whether the processed instruction's resulted in zero. Read Only. The flag is set to 0 when the ALU calculation is 1, and 0 when the output is non-zero.

OV Overflow Flag; Indicated whether the processed instruction's result overflowed the 32 bit space. Read only.
The overflow flag is set to the output of the carry out of the ALU.
With addition, this would set the bit to a 1 when true.
For subtraction, the inverse is true.

PML Permission Level; Inidcates the running process' permission level. Read only.

PML	Description
000	Low User Level
001	High User Level
010	Low Supervisor Level
011	High Supervisor Level
100	Low Hypervisor Level
101	High Hypervisor Level
110	Reserved
111	Top Hypervisor Level

The permission levels are explained in more detailed in their dedicated section.

PEMA Permission Accepted; Indictates whether a privaledged system call request was accepted. Read only

GINE Global Interrupt Enable; Indicates whether interrupts are enabled. Read/Write

SPCP Support CoProcessors; Indicates whether co-processor code is recognized as illegal instruction or is handled by an interrupt. (0 is illegal, 1 is interrupt handled) Read/Write.

The **STAT** register provides the programmer with various information about the current state of the processor.

In order to write to this register the bits that are defined as read only should be set to 0.

Optional In the event that the permission levels are not needed, they should be hard coded to 0x7, the highest permission level to avoid porting code. PEMA should also be hardcoded to a logic high for the same reasons stated.

OPCARP: Opcode Registration Table Pointer Register

31	0
Address	

The **OPCARP** register is a pointer for the Opcode Registration Table. This table exists to map co-processor opcodes to available opcodes in the processor. It is not necessary for the table to be hard coded, and this option is left up to the implementation.

Only permission levels higher than or equal to the High Supervisor Level can read or write to this register. If this register is not configured correctly before running an Operating System or bare-metal program, unintentional CPMI interrupts may occur with no proper service routine.

CPIDTP: Co-Processor ID Table Pointer Register

31	0
Address	

The **CPIDTP** register is a pointer for the Co-Processor ID Table. This table holds all co-processor IDs to allow the programmer to know how to handle software implementations of various co-processors.

All permission levels can read from the CPIDTP register. Only permission levels higher than the High Supervisor Level can write to this register, if it has not been hard coded.

UINTEN: Interrupt Enable Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	ROI	CPMI

The **UINTEN** register holds the settings for enabling various interrupts. More information about each interrupt is available in the interrupt section. The spots that are blank are usable by the implementation, while spots marked 'NA' are reserved for future use. Setting a bit field to 1 enables the interrupt vector, after ensuring that the GINE bit in **STAT** is set.

RPTINFO: Running Process Thread Info Register

31 16	15 8	7 0
PPID	CHPID	RESERVED

The **RPTINFO** register provides information about a process's threads. This information includes information for parent and child processes. The register fields are described in detail below.

PPID	Parent Process ID. Holds the parent's process ID.
CHPID	Child Process ID. The current thread number is stored here. 256 total threads are currently possible with using this scheme.
RESERVED	Reserved for future use.

All permission levels can read this register. Only permission levels equal to or higher than the High Supervisor Levels can write to this register.

RPID: Running Process ID Register

16 0
ID number

The RPID register holds the running process' ID number. This register is to be set by the kernel, and is not necessary otherwise. All permission levels can read from the register. Only permission levels equal to or higher than the High Supervisor Levels can write to this register.

3.3.2 Supervisor Registers

This section is still being developed. It will be updated in the future.

4 Permission Levels

The various permission levels are designed to allow for protected execution between user space, a kernel, and virtual environments. Permission levels are changed through dedicated instructions to increase or decrease the permission level counter. System calls can change the permission level in order to execute specific functions, but can be revoked in order to keep security. The individual function calls can be defined in the System Call Function Pointer Table (SYSFPT).

On boot, the default permission level is initialized to the highest permission level in order to properly configure the processor. There are no restrictions to access. However, after configuration the permission level should be lowered as required to ensure secure execution.

4.1 User Levels

4.1.1 Low User Level

User space processes are designed to run in this permission level. Write access to supervisor registers is revoked, and all modifying system calls are checked for allowed execution. Low priority, non time sensitive applications, high security risk applications should be run at this level. Since the Low User Level is the most restrictive priority level, in the event of a illegal operation by the application the processor should attempt to revert to a state before the illegal action and either kill or halt the offending process in order to prevent further illegal actions. Halting the process should be used to determine more information about the illegal action, while killing the process is ideal for more critical situations.

4.1.2 High User Level

This user level is designed to keep some security between user space and supervisor space, whiling loosening restrictions. Only reads are allowed to system and supervisor registers, and a limited number of system calls will not need to be verified. The verified system calls are specified in the 8 system call registers: SCPMHULX. Time critical user space processes, trusted system call saturated processes, and processes that require reading system and supervisor registers should use this permission level. registers.

4.2 Supervisor Levels

4.2.1 Low Supervisor Level

This permission level allows reads and writes to most system registers, with few restrictions. Hardware drivers, real time processes, and kernel tasks that do not require low level configuration should use this level.

4.2.2 High Supervisor Level

This permission level is for unrestricted access for operating system environments. Essential Kernel processes and bare-metal embedded applications are examples for using this permission level. All configuration registers are available, only hypervisor configuration and actions are restricted. If the hypervisor system is not implemented, this level is the highest permission level.

4.3 Hypervisor Levels

This section is still under development. At this time, the Hypervisor levels simply allow access to all functions of a processor, including the virtualization features. Since the virtualization functions are not defined yet, this is a placeholder for the time being.

5 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.

The Fusion-ISA uses a modified Harvard architecture, such that the programmer should have the appearance of a Harvard architecture. Compliance to this is not strict, as any variation in between is allowed, but the separation between data and instruction memory should be made.

5.1 Memory Locations for Vector Table

5.1.1 Interrupt Vector Table

The table below is the definition for the Interrupt Vectors. The vectors are padded by two addresses in order to give enough space for a call macro, which expands to two instructions. Additional interrupt vectors can be created, though the ones specified are necessary.

Address (32 bit)	Name	Definition
0x0000	Reset	Reset processor
0x0008	CPMI	Co-Processor Microcode Interrupt
0x0010	ROI	Register Overflow (Arithmetic)
0x0018	RESERVED	
0x0020	RESERVED	
0x0028	RESERVED	

Figure 4: Interrupt Vector Table

5.1.2 Exception List

Exception Code	Name	Definition
0x0000	INSFPEM	Insufficient Permissions
0x0001	MISADDR	Misaligned Address
0x0002	HALT	Halt Request
0x0003	PTERM	Terminate Process
0x0004	PKILL	Kill Process
0x0005	PEXIT	Process Exit (End)
0x0006	TTERM	Terminate Thread
0x0007	TKILL	Kill Thread
0x0008	TEXIT	Exit Thread (End)
0x0009	INVSYS	Invalid System Call
0x000a	INVPR	Invalid Permission Request

Figure 5: Exception List

6 Instruction Usage

This section will go over programming use of the instructions including the affects on the processor. Information on the instruction encodings and list of binary representations of each instruction is available in the 'Instructions' Section.

6.1 Integer

The following instructions operate on two registers, RSa and RSb, and store the result into Rd. The Z and OV flags are affected by the output of the instructions listed below. Overflows of register operations can be handled by the program, or the processor by checking the OV flag in the **STAT** register, or enabling the ROI interrupt. The Z flag is available as well to the programmer, but it does not have it's own interrupt to use. Since branch instructions exist to check if a register is equal to zero, the Z flag does not serve a purpose at this time.

Since arithmetic functions are simple to explain, the following table will denote the operands and function of the Integer instructions.

Integer Instructions

Instruction Name	Definition	Usage
add	Adds RSa and RSb, result stored in Rd	add $Rd, $RSa, $RSb
sub	Subtracts RSb from RSa, result stored in Rd	sub $Rd, $RSa, $RSb
addu	Adds RSa and RSb, result stored in Rd Does not affect OV flag	addu $Rd, $RSa, $RSb
subu	Subtracts RSb from RSa, result stored in Rd Does not affect OV flag	subu $Rd, $RSa, $RSb
not	Logically inverts RSa, result stored in Rd	not $Rd, $RSa
and	Logic AND operation on RSa and RSb, result stored in Rd	and $Rd, $RSa, $RSb
or	Logic OR operation on RSa and RSb, result stored in Rd	or $Rd, $RSa, $RSb
xor	Logic XOR operation on RSa and RSb, result stored in Rd	xor $Rd, $RSa, $RSb
sal	Arithmetic Shift RSa Left by the value of RSb, result stored in Rd	sal $Rd, $RSa, $RSb
sar	Arithmetic Shift RSa Right by the value of RSb, result stored in Rd	sar $Rd, $RSa, $RSb
sll	Logic Shift RSa Left by the value of RSb, result stored in Rd	sll $Rd, $RSa, $RSb
slr	Logic Shift RSa Right by the value of RSb, result stored in Rd	slr $Rd, $RSa, $RSb
comp	Compares the values of RSa and RSb. Sets Rd to 0x0 if equal, 1 if RSa is greater than RSb, and -1 if RSa is less than RSb	comp $Rd, $RSa, $RSb

6.2 Immediate

Similar to the Integer instructions, the Immediate instructions perform arithmetic operations on the RSa register and a 12 bit immediate. The table shown below describes the instructions available.

Immediate Instructions

Instruction Name	Definition	Usage
addi	Adds RSa and Immediate, result stored in Rd	addi $Rd, $RSa, 0xfff
subi	Subtracts Immediate from RSa, result stored in Rd	subi $Rd, $RSa, 0xfff
addui	Adds RSa and Immediate, result stored in Rd Does not affect OV flag	addui $Rd, $RSa, 0xfff
subui	Subtracts Immediate from RSa, result stored in Rd Does not affect OV flag	subui $Rd, $RSa, 0xfff
noti	Logically inverts Immediate, result stored in Rd	noti $Rd, 0xfff
andi	Logic AND operation on RSa and Immediate, result stored in Rd	andi $Rd, $RSa, 0xfff
ori	Logic OR operation on RSa and Immediate, result stored in Rd	ori $Rd, $RSa, 0xfff
xori	Logic XOR operation on RSa and Immediate, result stored in Rd	xori $Rd, $RSa, 0xfff
sali	Arithmetic Shift RSa Left by the Immediate value, result stored in Rd	sali $Rd, $RSa, 0xfff
sari	Arithmetic Shift RSa Right by the Immediate value, result stored in Rd	sari $Rd, $RSa, 0xfff
slli	Logic Shift RSa Left by the Immediate value, result stored in Rd	slli $Rd, $RSa, 0xfff
slri	Logic Shift RSa Right by the Immediate value, result stored in Rd	slri $Rd, $RSa, 0xfff
compi	Compares the RSa and the Immediate value. Sets Rd to 0x0 if equal, 1 if RSa is greater than the Immediate value, and -1 if RSa is less than the Immediate value	compi $Rd, $RSa, 0xfff

6.3 Load/Store

The Load Instructions feature a 14 bit immediate offset value, with a base address of the value of RSa. Since the architecture is big endian, the byte aligned address refers to the most significant byte. When reading values less than 32 bits wide, the upper bytes are set to zeros. For example, when loading a two byte value into Rd, the address in RSa and the next byte will be placed into the lower two bytes of Rd.

Load Instructions

Instruction Name	Definition	Usage
lw	Reads a 4 byte value from Memory at address RSa with a 14 bit Immediate offset, puts result in Rd	lw $Rd, 0x3fff($RSa)
lh	Reads a 2 byte value from Memory at address RSa with a 14 bit Immediate offset, puts result in Rd	lw $Rd, 0x3fff($RSa)
lb	Reads a 1 byte value from Memory at address RSa with a 14 bit Immediate offset, puts result in Rd	lw $Rd, 0x3fff($RSa)
lth	Reads a 3 byte value from Memory at address RSa with a 14 bit Immediate offset, puts result in Rd	lw $Rd, 0x3fff($RSa)

The Store Instructions are similar in nature to the Load instructions, with a 14 bit immediate value offset to the base address of $RSa. The value of $RSb will be written to the address calculated from the aforementioned sources. For writing values smaller than 32 bits wide, the lower bytes will be written in a big endian fashion to memory. For example, writing a two byte value to memory, would take the lower two bytes of $RSb, and write the higher byte to the address calculated and the lower byte to the address calculated plus 1.

Store Instructions

Instruction Name	Definition	Usage
sw	Writes the value of RSb at address RSa with a 14 bit Immediate offset.	lw $RSb, 0x3fff($RSa)
sh	Writes the lower two bytes of RSb at address RSa with a 14 bit Immediate offset.	lw $RSb, 0x3fff($RSa)
sb	Writes the least significant byte of RSb at address RSa with a 14 bit Immediate offset.	lw $RSb, 0x3fff($RSa)
sth	Writes the lower three bytes of RSb at address RSa with a 14 bit Immediate offset.	lw $RSb, 0x3fff($RSa)

The Load Immediate Instructions are for putting immediate values into registers. A 16 bit immediate value is stored into Rd, with variations on register file and location within the register. For unsigned upper immediate loads, only the upper 2 bytes are overwritten. This is to allow function calls to only take 2 instructions, and 1 register. All other load immediate instructions overwrite the entire register.

Load Immediate Instructions

Instruction Name	Definition	Usage
li	Put Immediate value into the lower two bytes of Rd in the GPREGF	li $Rd, 0xffff
lui	Put Immediate value into the upper two bytes of Rd in the GPREGF	lui $Rd, 0xffff
lni	Put unsigned Immediate value into the lower two bytes of Rd in the GPREGF	lni $Rd, 0xffff
luni	Put unsigned Immediate value into the upper two bytes of Rd in the GPREGF	luni $Rd, 0xffff
lgi	Put Immediate value into the lower two bytes of Rd in the GLBREGF	lgi $Rd, 0xffff
lugi	Put Immediate value into the upper two bytes of Rd in the GLBREGF	lugi $Rd, 0xffff
lngi	Put unsigned Immediate value into the lower two bytes of Rd in the GLBREGF	lngi $Rd, 0xffff
lungi	Put unsigned Immediate value into the upper two bytes of Rd in the GLBREGF	lungi $Rd, 0xffff
lsi	Put Immediate value into the lower two bytes of Rd in the SYSREGF	lsi $Rd, 0xffff
lusi	Put Immediate value into the upper two bytes of Rd in the SYSREGF	lusi $Rd, 0xffff
lnsi	Put unsigned Immediate value into the lower two bytes of Rd in the SYSREGF	lnsi $Rd, 0xffff
lunsi	Put unsigned Immediate value into the upper two bytes of Rd in the SYSREGF	lunsi $Rd, 0xffff

6.4 Branch/Jump

The Branch Instructions conditionally change the Program Counter based off of the operation of RSa and RSb. The 14 bit immediate field is PC relative to the address of the branch instruction.

Branch Instructions		
Instruction Name	Definition	Usage
beq	Branch to PC relative Immediate address when RSa and RSb are equal	beq $RSa, $RSb, 0x3fff
bne	Branch to PC relative Immediate address when RSa and RSb are not equal	bne $RSa, $RSb, 0x3fff
bgt	Branch to PC relative Immediate address when RSa is greater than RSb	bgt $RSa, $RSb, 0x3fff
blt	Branch to PC relative Immediate address when RSa is less than RSb	blt $RSa, $RSb, 0x3fff

---

The Jump Instructions change the Program Counter with a 21 bit immediate relative offset. The immediate is relative to the address of the jump instruction. Jump register instructions are absolute, and do not have a relative immediate. The value of RSa is used as the base address. Jump instructions that link save the return address into RA. The return address is the next 4 byte aligned address after the jump instruction.

Jump Instructions		
Instruction Name	Definition	Usage
j	PC relative jump	j 0x1fffff
jal	PC relative jump, save return address	jal 0x1fffff
jr	Jump register instruction, address calculated by immediate field plus RSa, save return address	jr 0x1fffff($RSa)
jrl	Jump register instruction, address calculated by immediate field plus RSa, save return address	jrl 0x1fffff($RSa)

6.5 System

The system instructions are more complex, but designed to utilize similar features to the rest of the main core or simplistic implementations of quick control sequences. Each instruction will get it's own paragraph due to the nature of the instructions not having a lot of similarities between them.

Name	Usage
syscall	syscall 0xff
sysret	sysret

The system call and system return instructions are used for their named purpose. The number used for the system calls are created in the System Call Table, which defines the address for system call routines. The table pointer should be created before a system call is made, otherwise a Invalid System Call exception will be raised. Permissions can be set for various system calls within the System Call Table, and the programmer should ensure that the permission level is adequate for using a system call. If permissions are adequate, the permission level will be raised appropriately and start executing the system call routine.

The sysret instruction should be placed at the end of the system call routine in order to switch back to the user space register file as well as lowering permissions.

Name	Usage
stspr	stspr $RSa, SPR
ldspr	ldspr $Rd, SPR

These instructions are for writing and reading to special purpose registers, respectively. The special purpose register address is put into the immediate field. Depending on the permission level setting of the registers, an Invalid Permission Level exception may be raised. Refer to the specific registers to determine if the permission level is adequate.

Name	Usage
sync	sync

The sync instruction is for ensuring memory synchronization. After the sync function is executed, the preceding instructions finish executing and all pending memory accesses to finish. The result allows a clean transition for context switches.

Name	Usage
lock	lock $Rd, $RSa
test	test $Rd, $RSa

The test and lock instructions provide exclusivity in multi-processor configurations, and between co-processors. The value in RSa is the address of the resource or memory location, and the success of the action is placed in Rd. If successful, the result will be zero. If unsuccessful, the resource that was attempted to be accessed will put an error code within Rd. The error code depends on the resource, and should be consulted in the respect location in the documentation of the co-processor or various other resource.

Name	Usage
pmir	pmir 0xff
pmd	test 0xff

These instructions are for increasing or decreasing the current permission level. The immediate field denotes the permission level that the processor should change to. In the event that a permission level cannot be changed, an Invalid Permission Level exception will be raised. Depending on the severity of the infraction, the running program may be terminated or halted. More information will be available in the permission level section.

6.6 Macros

The macro instructions defined here explain the usage and their expansions.

Name	Usage	Number of Instructions
call	call address_label	2
ret	ret	1
la	la $Rd, address_label	2

The call macro expands to a lui instruction and a jrl instruction with an immediate offset. The lower two bytes are used as the offset for the jump register, while the lui instruction loads the upper two bytes into register TMP0. TMP0 should be ensured to be cleared before using a call instruction, or an undefined sequence of events may occur. This register choice may change in the future to ensure better compatibility and resource usage, however register TMP0 should be avoided in use around call instructions as it may cause unintentional side effects.

The ret macro allows the program to return to the program from the subroutine. It expands into a jr instruction with no immediate offset. The RA register stores the return address, which the jr instruction uses.

The la macro loads the upper and lower bytes into the specified register, Rd. It expands into two instructions, li and lui. Currently there is no way to merge the two values into one register, however the macro will be updated soon to do so. It is advised against using this macro until further notice. The same functionality can be achieved with an or instruction and a li and lui instruction, however due to register usage this is not ideal to create as a macro.

---

7 Co-Processor Instruction Usage

This section is similar to the last, but pertains to instructions specific to various defined co-processors. At this time, no co-processors have been fully developed, nor verified, so this is left here as a placeholder for future revisions of the documentation.

7.1 Math Unit

7.2 Vector Operations Unit

7.3 Memory Management Unit

7.4 Inter-Processor Communications Unit

Part III

Instructions

8 Instruction Definitions

8.1 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 purely 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.

8.1.1 Integer

The integer 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	rd	rsa	rsb	shft	aluop
31 26	25 21	20 16	15 11	10 4	3 0

Overview: The Register/Integer Instruction Format is for 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:

Instruction	Description	ALU Operation (hex)
add	Add	0x0
sub	Subtract	0x1
addu	Add Unsigned	0x2
subu	Subtract Unsigned	0x3
not	Not	0x4
and	And	0x5
or	Or	0x6
xor	Exclusive Or	0x7
sal	Arithmetic Shift Left	0x8
sar	Arithmetic Shift Right	0x9
sll	Logic Shift Left	0xa
slr	Logic Shift Right	0xb
comp	Compare	0xc
Reserved	n/a	0xd
Reserved	n/a	0xe
Reserved	n/a	0xf

8.1.2 Immediate

Immediate Instruction Format

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

Overview: The Immediate Instruction Format is for ALU operations that require an immediate value. The immediate field is only 12 bits wide, so in the event that a larger value is required, the Load Immediate Format should be used. There is only a single source register, RSa, with the other source being the immediate value. The result is stored into register Rd.

The following instructions use this encoding:

Instruction	Description	ALU Operation (hex)
addi	Add Immediate	0x0
subi	Subtract Immediate	0x1
addui	Add Unsigned Immediate	0x2
subui	Subtract Immediate	0x3
noti	Invert Immediate	0x4
andi	And Immediate	0x5
ori	Or Immediate	0x6
xori	Exclusive Or Immediate	0x7
sali	Arithmetic Shift Left	0x8
sari	Arithmetic Shift Right	0x9
slli	Logic Shift Left	0xa
slri	Logic Shift Right	0xb
compi	Compare Immediate	0xc

8.1.3 Load/Store

Load Instruction Format

opcode		rd		rsa		funct		Immediate	
31	26	25	21	20	16	15	14	13	0

Overview: The Load Instruction Format is for reading values from memory into a register. A 14 bit immediate is used for a relative address calculation with RSa as the base address. The value is then stored into register Rd. The funct field is used to determine the byte size to read from memory. This encoding is described in the table below.

Funct	Description
00	1 word (32 bits)
01	half word (16 bits, upper)
10	24 bits, upper
11	byte, upper

The following instructions use this encoding:

Instruction	Description	Funct
lw	Load Word	0x0
lh	Load Half Word	0x1
lb	Load Byte	0x3
lth	Load Three (Bytes, 24 bits)	0x2

Load Immediate Instruction Format

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

Overview: The Load Immediate Instruction Format is for loading immediate values into a register. The DSEL value determines the destination and function of the instruction. This is to allow the selection of various register files to be updated. For unsigned upper immediate loads, only the upper 2 bytes are overwritten. This is to allow function calls to only take 2 instructions, and 1 register. All other load immediate instructions overwrite the entire register. The DSEL bit values and their function are shown below. The MSB of DSEL determines if the value is unsigned (1) or signed (0).

DSEL (bin)	Destination
0000	General purpose register file
0001	System Register File
0010	Global Co-Processor Register File
0100	General purpose register file (upper 16 bits)
0101	System Register File (upper 16 bits)
0110	Global Co-Processor Register File (upper 16 bits)
1000	General purpose register file (Unsigned)
1001	System Register File (Unsigned)
1010	Global Co-Processor Register File (Unsigned)
1100	General purpose register file (Unsigned, upper 16 bits)
1101	System Register File (Unsigned, upper 16 bits)
1110	Global Co-Processor Register File (Unsigned, upper 16 bits)

The following instructions use this encoding:

Instruction	Description	DSEL
li	load immediate (gprf)	0x0
lsi	load immediate system	0x1
lgi	load immediate global	0x2
lui	load immediate (gprf) (upper)	0x4
lusi	load immediate system (upper)	0x5
lugi	load immediate global (upper)	0x6
lni	load immediate (gprf) (unsigned)	0x8
lnsi	load immediate system (unsigned)	0x9
lngi	load immediate global (unsigned)	0xa
luni	load immediate (gprf) (unsigned,upper)	0xc
lunsi	load immediate system (unsigned,upper	0xd
lungi	load immediate global (unsigned,upper)	0xe

Store Instruction Format

opcode		Funct		Immediate[13:11]		rsa		rsb		Immediate[10:0]	
31	26	25	24	23	21	20	16	15	11	10	0

Overview: The Store Instruction Format is for writing register values to memory. A 14 bit immediate is used for a relative address calculation with RSa as the base address. The value in RSb is then written to memory. The funct field is used to determine the byte size to write to memory. This encoding is described in the table below.

Funct	Description
00	1 word (32 bits)
01	half word (16 bits, upper)
10	24 bits, upper
11	byte, upper

The following instructions use this encoding:

Instruction	Description	Funct
sw	Store Word	0x0
sh	Store Half Word	0x1
sb	Store Byte	0x3
sth	Store Three (Bytes, 24 bits)	0x2

8.1.4 Branch/Jump

Jump Instruction Format

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

Overview: The Jump Instruction format is for changing the Program Counter. All Jumps are relative, except Jump Register Instructions, due to the non byte aligned size (20 bits) of the immediate field. Jumps use the value in register RSa to calculate the new PC. For pure relative jumps, the Zero register is used as the base. For jump instructions that link the program counter, the PC is stored in RA0, or R4 in the General Purpose Register file.

The following instructions use this encoding:

Instruction	Description
j	Jump
jal	Jump and Link
jr	Jump Register
jrl	Jump Register and Link

Branch Instruction Format

opcode		Immediate[13:9]		rsa		rsb		Immediate[8:0]		funct	
31	26	25	21	20	16	15	11	10	2	1	0

Overview: The Branch Instruction Format is used for conditionally changing the Program Counter. The PC is only updated with whichever operation is indicated as being true or false.

The following instructions use this encoding:

Instruction	Description	Funct
beq	Branch if Equal	00
bne	Branch if not Equal	01
bgt	Branch if Greater than	10
blt	Branch if Less than	11

8.1.5 System Instructions

System Instruction Format

opcode		rd		rsa		Funct		Immediate[7:0]	
31	26	25	21	20	16	15	8	7	0

Overview: The System Instruction Format is used for various system controls. Their functions vary, and are described in more detail in their individual listings. These instructions are not required, but recommended for implementation. While technically the System Instructions are in the co-processor opcode space, they have a designated opcode of 110000 as shown in the OPCodes section.

The following instructions use this encoding:

Instruction	Description	Funct	Uses Immediate
syscall	System Call	0x00	yes
sysret	System Return	0x01	no
stspr	Store Special Purpose Register	0x02	yes
ldspr	Load Special Purpose Register	0x03	yes
sync	Synchronize memory/Flush Pipeline	0x04	no
lock	Lock Memory	0x05	yes
test	Test Lock	0x06	yes
pmir	Permission Increase Request	0x07	yes
pmd	Permission Decrease	0x08	yes

8.1.6 Co-Processor

Since the co-processors 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 co-processor ID number. More information about co-processors can be accessed in the Co-Processor section.

8.2 Opcodes

The opcodes are designed to provide a simple decode unit for the processor. The figure below shows the breakdown within the opcode.

CPEN	Mem/PC[1:0]	Reg Operands[2:0]
5	4 3	2 0

The most significant bit of the opcode field determines if the instruction is a Co-processor instruction or not. This is to offload decoding of co-processor instructions for the individual co-processors, and not cause any requirements for future developments of co-processor instruction decoding.

Bits 3 and 4 of the opcode field are for determining memory access, changing the program counter, and linking the return address to register RA. A possible way to generate signals for decoding the opcode is provided below in verilog.

```
pc_change = ~opcode[4]
pc_link = opcode[4] ~| opcode[3]
mem_access = opcode[4] & opcode[3]
```

The lower 3 bits of the opcode field determine the operand usage of an instruction, including Rd, RSa, RSb, and immediates. A possible way to generate signals for decoding the opcode is provided below in verilog.

```
use_rd = ( opcode[1] | opcode[0] ) & ( ~opcode[2] | opcode[1] )
use_rsa = opcode[2] | opcode[1]
use_rsb = ( opcode[2] & opcode[0] ) | ( opcode[1] & opcode[0] )
use_immediate = opcode[2] | ~opcode[1]
```

8.2.1 List of OPCodes

Op Code	Description
010011	Register ALU instructions
010000	Load Immediate instructions
010110	Immediate ALU instructions
001101	Branch Instructions
001100	Jump Instructions
000100	Jump and Link Instructions
110000	System Instructions
011110	Load Instructions
011101	Store Instructions
1XXXXX	Co-Processor Instructions

Each bit of the opcode directly affects the resources required for each instruction. This choice was made in hopes of reducing decode complexity. The MSB of the opcode selects if the instruction is for the co-processor, or main core. This makes it incredibly easy to determine

which core the instruction goes to, without sacrificing more opcode bits. The LSB of the opcode selects ALU usage, the second to last bit selects the immediate field.

However, in some cases the more significant bit will modify the functions of the later bits. The 3rd to last bit in the opcode selects a change in PC, but the following bits do not correspond to the previous meanings mentioned. The 4th bit denotes a system instruction, and the 5th bit denotes memory access.

8.3 List of Instructions

Instruction Instruction Summary Table

Integer Instructions		
Instruction	Function	Binary
add	Rd = RSa + RSb	010011dddddaaaaabbbbbxxxxxxx0000
sub	Rd = RSa - RSb	010011dddddaaaaabbbbbxxxxxxx0001
addu	Rd = RSa + RSb (Unsigned)	010011dddddaaaaabbbbbxxxxxxx0010
subu	Rd = RSa - RSb (Unsigned)	010011dddddaaaaabbbbbxxxxxxx0011
not	Rd = !RSa	010011dddddaaaaaxxxxxxxxxxxx0100
and	Rd = RSa & RSb	010011dddddaaaaabbbbbxxxxxxx0101
or	Rd = RSa | RSb	010011dddddaaaaabbbbbxxxxxxx0110
xor	Rd = RSa $\oplus$ RSb	010011dddddaaaaabbbbbxxxxxxx0111
sal	Rd = RSa $\ll$ RSb	010011dddddaaaaabbbbbsssssss1000
sar	Rd = RSa $\gg$ RSb	010011dddddaaaaabbbbbsssssss1001
sll	Rd = RSa $\ll$ RSb	010011dddddaaaaabbbbbsssssss1010
slr	Rd = RSa $\ggg$ RSb	010011dddddaaaaabbbbbsssssss1011
comp	Rd = (RSa == RSb);(RSa $>$ RSb);(RSa $<$ RSb)	010011dddddaaaaaxxxxxsssssss1100
Immediate Instructions		
addi	Rd = RSa + Imm	010110dddddaaaaaiiiiiiiiiiii0000
subi	Rd = RSa - Imm	010110dddddaaaaaiiiiiiiiiiii0001
addui	Rd = RSa + Imm (Unsigned)	010110dddddaaaaaiiiiiiiiiiii0010
subui	Rd = RSa - Imm (Unsigned)	010110dddddaaaaaiiiiiiiiiiii0011
noti	Rd = ! Imm	010110ddddd00000iiiiiiiiiiii0100
andi	Rd = RSa & Imm	010110dddddaaaaaiiiiiiiiiiii0101
ori	Rd = RSa | Imm	010110dddddaaaaaiiiiiiiiiiii0110
xori	Rd = RSa $\oplus$ Imm	010110dddddaaaaaiiiiiiiiiiii0111
sali	Rd = RSa $\ll$ Imm	010110dddddaaaaaiiiiiiiiiiii1000
sari	Rd = RSa $\gg$ Imm	010110dddddaaaaaiiiiiiiiiiii1001
slli	Rd = RSa $\ll$ Imm	010110dddddaaaaaiiiiiiiiiiii1010
slri	Rd = RSa $\ggg$ Imm	010110dddddaaaaaiiiiiiiiiiii1011
compi	Rd = (RSa == Imm);(RSa $>$ Imm);(RSa $<$ Imm)	010110dddddaaaaaiiiiiiiiiiii1100
Load Instructions		
lw	Rd <- Imm(RSa)	011110dddddaaaaa00iiiiiiiiiiiiii
lh	Rd <- Imm(RSa)	011110dddddaaaaa01iiiiiiiiiiiiii
lb	Rd <- Imm(RSa)	011110dddddaaaaa11iiiiiiiiiiiiii
lth	Rd <- Imm(RSa)	011110dddddaaaaa10iiiiiiiiiiiiii
Load Immediate Instructions		
li	(GPREGF) Rd = Imm	010000ddddd0000iiiiiiiiiiiiiiii
lsi	(SYSREGF) Rd = Imm	010000ddddd0001iiiiiiiiiiiiiiii
lgi	(GLBREGF) Rd = Imm	010000ddddd0010iiiiiiiiiiiiiiii

Load Immediate Instructions (cont.)		
Instruction	Function	Binary
lui	(GPREGF) Rd = Imm (upper 16 bits)	010000ddddd0011iiiiiiiiiiiiiiii
lusi	(SYSREGF) Rd = Imm (upper 16 bits)	010000ddddd0100iiiiiiiiiiiiiiii
lugi	(GLBREGF) Rd = Imm (upper 16 bits)	010000ddddd0101iiiiiiiiiiiiiiii
lni	(GPREGF) Rd = Imm (unsigned)	010000ddddd1000iiiiiiiiiiiiiiii
lnsi	(SYSREGF) Rd = Imm (unsigned)	010000ddddd1001iiiiiiiiiiiiiiii
lngi	(GLBREGF) Rd = Imm (unsigned)	010000ddddd1010iiiiiiiiiiiiiiii
luni	(GPREGF) Rd = Imm (upper 16, unsigned)	010000ddddd1011iiiiiiiiiiiiiiii
lunsi	(SYSREGF) Rd = Imm (upper 16, unsigned)	010000ddddd1100iiiiiiiiiiiiiiii
lungi	(GLBREGF) Rd = Imm (upper 16, unsigned)	010000ddddd1101iiiiiiiiiiiiiiii
Store Instructions		
sw	RSb -> Imm(RSa)	01110100iiiaaaaabbbbbiiiiiiiiii
sh	RSb -> Imm(RSa)	01110100iiiaaaaabbbbbiiiiiiiiii
sb	RSb -> Imm(RSa)	01110111iiiaaaaabbbbbiiiiiiiiii
sth	RSb -> Imm(RSa)	01110110iiiaaaaabbbbbiiiiiiiiii
Jump Instructions		
j	Next PC <- (R0 + Imm)	001100iiiii00000iiiiiiiiiiiiiiii
jal	Next PC <- (R0 + Imm); RA0 <- PC	000100iiiii00000iiiiiiiiiiiiiiii
jr	Next PC <- (RSa + Imm)	001100iiiiiaaaaaiiiiiiiiiiiiiiii
jrl	Next PC <- (RSa + Imm); RA0 <- PC	000100iiiiiaaaaaiiiiiiiiiiiiiiii
Branch Instructions		
beq	Next PC <- (RSa == RSb) ? PC+Imm : PC+4	001101iiiiiaaaaabbbbbiiiiiiiii00
bne	Next PC <- (RSa != RSb) ? PC+Imm : PC+4	001101iiiiiaaaaabbbbbiiiiiiiii01
bgt	Next PC <- (RSa > RSb) ? PC+Imm : PC+4	001101iiiiiaaaaabbbbbiiiiiiiii10
blt	Next PC <- (RSa < RSb) ? PC+Imm : PC+4	001101iiiiiaaaaabbbbbiiiiiiiii11
System Instructions		
syscall	System Call (Raise Privilege)	11000ddddd aaaaaa 00000000 iiiiiiii
sysret	System Return (Lower Privilege)	11000ddddd aaaaaa 00000001 iiiiiiii
stspr	(SYSREGF) Rd <- RSa	11000ddddd aaaaaa 00000010 xxxxxxxx
ldspr	(SYSREGF) RSa -> Rd	11000ddddd aaaaaa 00000011 xxxxxxxx
sync	Flush Pipeline	11000xxxxxxxxxx 00000100 xxxxxxxx
pmir	PML + Imm ?	11000xxxxxxxxxx 00000111 iiiiiiii
pmd	PML - Imm ?	11000xxxxxxxxxx 00001000 iiiiiiii

8.3.1 Co-Processor

More information on co-processor instructions will be available in the future. This section is still under development.

9 Exceptions and Interrupts

This ISA differentiates between Exceptions and Interrupts based on function. Interrupts are vectored, to allow the running process to define what should be done. Exceptions are more based in hardware, and require minimal setup in software.

9.1 Exceptions

This section is still under development.

9.1.1 User Level

9.1.2 Supervisor Level

9.2 Interrupts

This section is still under development.

9.2.1 User Level

9.2.2 Supervisor Level

Part IV

Co-Processors

10 Co-Processor Overview

Co-processors are the main point of the Fusion-Core architecture. As the ISA strictly specifies the main core for code compatibility, the implementation is free to use whichever co-processors that would be necessary for an application. As speeding up the common case is the main goal in CPU design, these co-processors should enhance the common case for the specific implementation. The ISA has no specific restrictions for a co-processor, but for non-memory mapped co-processors, they should be created as dictated by the interface specified below.

Examples of co-processors could be hardware acceleration for vector instructions, encryption, floating point, or communication. The co-processor can be as complex, or as simple as one requires. There is no limitation for what kind of co-processor that could be used, only the number of co-processors that can utilize the opcode space allocated for co-processors. The co-processors can also share the same opcode, which can help create vectorized instruction units such as GPU like co-processors, coupled directly to the main core's instruction flow. The programmer would be able to exploit more locality in programming as switching to different sections of memory is not required. It should be noted that this last sentence is not a requirement, but just a possibility and the current implementation of the default case for the GNU Binutils port. This can easily be modified with a custom linker script.

The reasoning behind co-processors use is to allow for code compatibility between nearly all implementations of the Fusion-Core ISA, and allowing for an enhancement to the common case of a processor's use. Most architectures do not allow for radical implementations in the architecture itself, only the micro-architecture. And by keeping complex instructions away from the main core, a simple decode unit can be created to allow for potentially faster pipelining.

10.1 Co-Processor Interface

Co-Processor Opcode space Co-processor instructions are enabled through use of the most significant bit of the opcode field. The processor should be able to distinguish between main core and co-processor instructions quickly, and a single bit is the simplest way of doing so. The other 5 bits are completely usable for whichever purpose the implementation could desire.

In order to let the programmer know what co-processors are available, the opcode registration table should contain the co-processor ID (CPID). This ID is provided by the Fusion-Core developers. Please send an email to cpid@fusion-core.org for making requests for a new co-processor. This ID will be implemented to allow for proper disassembly and assembly of the instructions in the GNU Binutils port. It is also possible to not use an official CPID, however no support will be given to the developers of the co-processor.

11 Global Register File

The Global Register File is for simple message passing, and creating locks between co-processors. As it may be necessary to wait for a value to be computed by a co-processor, or lock specific parts of memory, the Global Register File creates an interface for ease of use between processing units.

This section is under development and will be updated to explain the connections and registers available.

12 Recommended Co-Processors

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

12.1 Math Unit

12.1.1 Registers

12.1.2 Instructions

12.2 System Unit

12.2.1 Registers

12.2.2 Instructions

12.3 Memory Management Unit

12.3.1 Registers

12.3.2 Instructions

12.4 Inter-Processor Communications Unit

12.4.1 Registers

12.4.2 Instructions