Fusion-Core ISA Definition: Revision 0.1

Dylan Wadler

December 29, 2017

Contents

Part I

Administrative

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 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.

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	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

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

Figure 2: Special Registers

System Registers			
Register Name	Description	Width (bytes)	Address (Hex)
CPUREV	CPU Revision	1	0x00000000
CPUNUM	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
HTINFO	Hardware Thread Info	0x00000008
HTCTL	Hardware Thread Control	0x0000000c

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.

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 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.

3.4 Special Purpose Registers

3.4.1 System Registers

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 0, and 1 when the output it 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 systemcall request was accepted. Read o

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

SPCP Support CoProcessors; Indicates whether co-processor code is recognized as illegal instruction or microcode operation.
Read/Write.

In order to write to this register, the bits that are read only can be set to any value.

Optionality 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.

3.4.2 Supervisor Registers

4 Permission Levels

4.1 User Levels

4.1.1 Low User Level

User space programs are designed to run in this permission level. Write access to supervisor registers is revoked.

4.1.2 High User Level

4.2 Supervisor Levels

4.2.1 Low Supervisor Level

4.2.2 High Supervisor Level

4.3 Hypervisor Levels

4.3.1 Low Hypervisor Level

4.3.2 High Hypervisor Level

4.3.3 Top Hypervisor Level

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.

5.1 Memory Locations for Vector Table

5.1.1 Interrupt Vector Table

5.1.2 Exception Vector Table

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

Figure 4: Exception Vector Table

Part III
Instructions

6 Instruction Definitions

6.1 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.

6.1.1 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 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
tcmp	2's Complement	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

6.1.2 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 Immediate	0x3
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 Immedaite	0xc

6.1.3 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 Instrution Format

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

Overview: The Load Immediate Instruction Format is for loading immeidate 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. 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
0011	General purpose register file (upper 16 bits)
0100	System Register File (upper 16 bits)
0101	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)
1011	General purpose register file (Unsigned, upper 16 bits)
1100	System Register File (Unsigned, upper 16 bits)
1101	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)	0x3
lusi	load immediate system (upper)	0x4
lugi	load immediate global (upper)	0x5
lni	load immediate (gprf) (unsigned)	0x8
lnsi	load immediate system (unsigned)	0x9
lngi	load immediate global (unsigned)	0xa
luni	load immediate (gprf) (unsigned,upper)	0xb
lunsi	load immediate system (unsigned,upper	0xc
lungi	load immediate global (unsigned,upper)	0xd

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

6.1.4 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

6.1.5 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.

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	Syncronize 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

6.1.6 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.

6.2 List of OPCodes

Op Code	Description	Instrucion Formats

6.3 Instruction Details

This section will go into detail about the instructions that are available, with their functions and options.

6.3.1 Integer

6.4 List of Instructions

Instruction Instruction Summary Table

Integer Instructions		
Instruction	Function	Binary
add	Rd = RSa + RSb	000001dddddaaaaabbbbbxxxxxxx0000
sub	Rd = RSa - RSb	000001dddddaaaaabbbbbxxxxxxx0001
addu	Rd = RSa + RSb (Unsigned)	000001dddddaaaaabbbbbxxxxxxx0010
subu	Rd = RSa - RSb (Unsigned)	000001dddddaaaaabbbbbxxxxxxx0011
tcmp	Rd = !Rsa	000001dddddaaaaaxxxxxxxxxxxx0100
and	Rd = RSa & RSb	000001dddddaaaaabbbbbxxxxxxx0101
or	Rd = RSa $\mid$ RSb	000001dddddaaaaabbbbbxxxxxxx0110
xor	Rd = RSa $\oplus$ RSb	000001dddddaaaaabbbbbxxxxxxx0111
sal	Rd = RSa $\ll$ RSb	000001dddddaaaaabbbbbsssssss1000
sar	Rd = RSa $\gg$ RSb	000001dddddaaaaabbbbbsssssss1001
sll	Rd = RSa $\ll$ RSb	000001dddddaaaaabbbbbsssssss1010
slr	Rd = RSa $\ggg$ RSb	000001dddddaaaaabbbbbsssssss1011
comp	Rd = (RSa == RSb);(RSa $>$ RSb);(RSa $<$ Rsb)	000001dddddaaaaaxxxxxsssssss1100
Immediate Instructions		
addi	Rd = RSa + Imm	000011dddddaaaaaiiiiiiiiiiii0000
subi	Rd = RSa - Imm	000011dddddaaaaaiiiiiiiiiiii0001
addui	Rd = RSa + Imm (Unsigned)	000011dddddaaaaaiiiiiiiiiiii0010
subui	Rd = RSa - Imm (Unsigned)	000011dddddaaaaaiiiiiiiiiiii0011
andi	Rd = RSa & Imm	000011dddddaaaaaiiiiiiiiiiii0101
ori	Rd = RSa $\mid$ Imm	000011dddddaaaaaiiiiiiiiiiii0110
xori	Rd = RSa $\oplus$ Imm	000011dddddaaaaaiiiiiiiiiiii0111
sali	Rd = RSa $\ll$ Imm	000011dddddaaaaaiiiiiiiiiiii1000
sari	Rd = RSa $\gg$ Imm	000011dddddaaaaaiiiiiiiiiiii1001
slli	Rd = RSa $\ll$ Imm	000011dddddaaaaaiiiiiiiiiiii1010
slri	Rd = RSa $\ggg$ Imm	000011dddddaaaaaiiiiiiiiiiii1011
compi	Rd = (RSa == Imm);(RSa $>$ Imm);(RSa $<$ Imm)	000011dddddaaaaaiiiiiiiiiiii1100
Load Instructions		
lw	Rd $<$- Imm(RSa)	010010dddddaaaaa00iiiiiiiiiiiii
lh	Rd $<$- Imm(RSa)	010010dddddaaaaa01iiiiiiiiiiiii
lb	Rd $<$- Imm(RSa)	010010dddddaaaaa11iiiiiiiiiiiii
lth	Rd $<$- Imm(RSa)	010010dddddaaaaa10iiiiiiiiiiiii
Load Immediate Instructions		
li	(GPREGF) Rd = Imm	000010ddddd0000iiiiiiiiiiiiiiii
lsi	(SYSREGF) Rd = Imm	000010ddddd0001iiiiiiiiiiiiiiii
lgi	(GLREGF) Rd = Imm	000010ddddd0010iiiiiiiiiiiiiiii

Instruction	Function	Binary
lui	(GPREGF) Rd = Imm (upper 16 bits)	000010ddddd0011iiiiiiiiiiiiiiii
lusi	(SYSREGF) Rd = Imm (upper 16 bits)	000010ddddd0100iiiiiiiiiiiiiiii
lugi	(GLREGF) Rd = Imm (upper 16 bits)	000010ddddd0101iiiiiiiiiiiiiiii
lni	(GPREGF) Rd = Imm (unsigned)	000010ddddd1000iiiiiiiiiiiiiiii
lnsi	(SYSREGF) Rd = Imm (unsigned)	000010ddddd1001iiiiiiiiiiiiiiii
lngi	(GLREGF) Rd = Imm (unsigned)	000010ddddd1010iiiiiiiiiiiiiiii
luni	(GPREGF) Rd = Imm (upper 16, unsigned)	000010ddddd1011iiiiiiiiiiiiiiii
lunsi	(SYSREGF) Rd = Imm (upper 16, unsigned)	000010ddddd1100iiiiiiiiiiiiiiii
lungi	(GLREGF) Rd = Imm (upper 16, unsigned)	000010ddddd1101iiiiiiiiiiiiiiii
Store Instructions		
sw	RSb -> Imm(RSa)	01101000iiiaaaaabbbbbiiiiiiiiiii
sh	RSb -> Imm(RSa)	01101001iiiaaaaabbbbbiiiiiiiiiii
sb	RSb -> Imm(RSa)	01101011iiiaaaaabbbbbiiiiiiiiiii
sth	RSb -> Imm(RSa)	01101010iiiaaaaabbbbbiiiiiiiiiii
Jump Instructions		
j	Next PC <- (R0 + Imm)	000110iiiiiaaaaaiiiiiiiiiiiiiiii
jal	Next PC <- (R0 + Imm); RA0 <- PC	000111iiiiiaaaaaiiiiiiiiiiiiiiii
jr	Next PC <- (RSa + Imm)	000110iiiiiaaaaaiiiiiiiiiiiiiiii
jrl	Next PC <- (RSa + Imm); RA0 <- PC	000111iiiiiaaaaaiiiiiiiiiiiiiiii
Branch Instructions		
beq	Next PC <- (RSa == RSb) ? PC+Imm : PC+4	000101iiiiiaaaaabbbbbiiiiiiiii00
bne	Next PC <- (RSa != RSb) ? PC+Imm : PC+4	000101iiiiiaaaaabbbbbiiiiiiiii01
bgt	Next PC <- (RSa > RSb) ? PC+Imm : PC+4	000101iiiiiaaaaabbbbbiiiiiiiii10
blt	Next PC <- (RSa < RSb) ? PC+Imm : PC+4	000101iiiiiaaaaabbbbbiiiiiiiii11
System Instructions		
syscall	System Call (Raise Privalege)	00100dddddaaaaaa00000000iiiiiiii
sysret	System Return (Lower Privalege)	00100dddddaaaaaa00000001iiiiiiii
stspr	(SYSRF) Rd <- RSa	00100dddddaaaaaa00000010xxxxxxxx
ldspr	(SYSRF) RSa -> Rd	00100dddddaaaaaa00000011xxxxxxxx
sync	Flush Pipeline	00100xxxxxxxxxxx00000100xxxxxxxx
pmir	PML + Imm ?	00100xxxxxxxxxxx00000111iiiiiiii
pmd	PML - Imm ?	00100xxxxxxxxxxx00001000iiiiiiii

6.4.1 Immediate

6.4.2 Load/Store

6.4.3 Branch/Jump

6.4.4 Co-Processor

7 Exceptions and Interrupts

7.1 Exceptions

7.2 Interrupts

7.2.1 User Level

7.2.2 Supervisor Level

Part IV

Co-Processors

8 Co-Processor Overview

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.

8.1 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.

8.1.1 Co-Processor Conventions

8.1.2 Register Connections

8.1.3 Decode unit Connections

8.2 Interface Connection Definitions

8.3 Adding custom Co-Processor

8.4 List of Co-Processors

8.4.1 Floating Point

8.4.2 System Unit

8.4.3 Memory Management Unit

8.4.4 Multiprocessor Communication Unit

9 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.

10 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.

10.1 Floating Point Unit

10.1.1 Registers

10.1.2 Instructions

10.2 System Unit

10.2.1 Registers

10.2.2 Instructions

10.3 Memory Management Unit

10.3.1 Registers

10.3.2 Instructions

10.4 Multiprocessor Communication Unit

10.4.1 Registers

10.4.2 Instructions