Scheduled Dataflow Architecture:
Instruction Set Reference 2.0.0

May 22, 2006

The purpose of this document is to provide a self-contained reference that could be useful both for the design of the Scheduled Dataflow Architecture and the design of a compiler targeting this architecture.

1 Conceptual Vision of the Machine

The Scheduled Dataflow Architecture consists of the following four main building blocks:

- the *Instruction and Frame Memory (Instruction add Data Cache)*
- the *Global Memory*
- the *Global Registers*
- the *Per Thread Register Contexts*
- the *Execution Processor (EPs)*
- the *Synchronization Processor (SPs)*
- the *Scheduling Unit*

1.1 The Instruction and Frame Memory (Instruction Caches)

The Instruction and Frame Memory is a local memory of the machine. The implemetation may use multiple Instruction caches for different clusters.
The continuation for SDF threads describe a local memory for the thread called a Frame. A fixed sized memory is allocated to a threa upon its creation. The inputs to the created thread are stored in its Frame. One or more Frame caches can be used in an actual implemetnation to minimize contention for cache accesses.

1.2 The Global Memory

The global memory can be used for data that is shared among threads. Multiple semantics can be applied to the global memory. Conventional memory access will represented by READ and WRITE instructions. I-structure sementics can be applied by using IFETCH and ISTORE instructions. The I-Structure Memory guarantees the synchronization *among* data accesses by different processors. In a future implementation, one may consider other semantics such as the J and L structure. Cache memory for the global data con be used to improve access time.
Global memory, regardless of the semantics being applied will be acessed using our R format. The address is defined using two registers. One register contains the base address while the second register contains an offset. If R0 (Register0) of the per thred register set, which is permanently hardwired to zero) as offset, the address mode becomes equivalent to an absolute address mode.

1.3 Global Registers

There are 32 global registers that can be accessed by SPs and EPs. These registers will be labeled as G registers. The global register set is mainly used as scratch space. There are instructions to support the transfer data between global register set and per thread register context. Simple arithmetic operations like addition, subtraction and etc.. are supported. Instructions involving global registers are prefixed with G (GADD, GSUB, ...).

1.4 Per Thread Register Contexts

SDF provides both Integer and Floating point registers for each active or executing thread. Integer registers with a context will be labeled as R registers while the floating point registers will be labeled as F registers. Instructions for the floating point registers will include F as prefix (FADD, FSUB, ...). Operation on R registers will not involve any prefix (ADD, SUB, ...). Register sets can be accessed by both the SPs and EPs. In one implemetation, we use 32 floating point and 64 integer registers per context.

1.5 The Execution Processor (EP)

The Execution Processor includes:

- the Execution Pipeline (XP)
- the Program Counter (PC)
- the Running Context Pointer (RCP)
- 16 contexts (CTX00, CTX01, . . .), also accessible by SP
- an Instruction cache

1.5.1 The Running Context

RCP always point to the running context. The running context includes a set of register as specified in the Instruction Set Section.
Active contexts are those contexts that have been allocated by some execution thread but that have not been filled out with data from Frame Memory or that are completing the storing of data into the Frame Memory.
The Execution Processor always has one running context, a number of active contexts, and a number of unallocated contexts.

1.5.2 The Execution Pipeline

The Execution Pipeline consists of four stages, which are ordered as follows:

- Instruction Fetch
- Decode and Operand Fetch (up to 2 operands may be fetched)
- Execute
- Write Back (up to 1 operands may be stored)

All operations involving operands (Operand Fetch and Write Back) act exclusively on the Running Context registers in a non-blocking fashion. The Synchronization Processor takes care of loading and storing data from/to Operand (Frame) Cache/Memory as specified in the following SubSection.
The execution of a code block can start only once the SP has loaded all the values that are needed by the frame associated with that code block.

1.6 The Synchronization Processor (SP)

The Synchronization Processor takes care of loading and storing operands in the active contexts. The active contexts are all allocated contexts, except the running context, which contain operand to be stored to or retrieved from the Operand (Frame) Cache+Memory.
The SP contains also:

- an Operand (Frame) cache
- an I-Structure and general memory

The SP can access the register contexts in the EP. SP and EP also need to exchange <PC,RCP> (Thread-ID). Further details will be explained in the thread management Section.

2 Instruction Set

2.1 Registers

The machine supports multiple contexts. Each context has 32 integer register pairs and 32 floating registers. Each register of a pair can be addressed separately. Integer register should have enough room to accommodate all possible 3 basic types, which are: Boolean, Integer, Character. And floating point register can accomodate 32 single precision floating point value or 16 double precision floating point value. Register R0 is hardwired to 0.
The machine must guarantee at least the following data ranges for the previous types.

TYPE	RANGE
Boolean	TRUE,FALSE
Character	0..255
Integer	-2147483648 .. 2147483647 (-2^{31} .. $2^{31} - 1$)
Real	32-bit single-precision or 64-bit double-precision (IEEE 754 standard))

2.2 Notation

- RD indicates a destination register.
- RS indicates a source register.
- I indicates an I-Structure; F indicates a Frame; C indicates a code-block; D indicates an I/O device
- <I, indx> indicates the I-Structure entry $I[indx]$
- ≪F, offset≫ indicates the Frame data at offset 'offset' in frame F
- '&' means 'address of' when placed before one of the previous objects

2.3 Instruction Formats

- **R format** (Register to Register operations)

R	OpCode	RS1	RS2	RD	RESERVED
0 1	2 7	8 13	14 19	20 24	25 31

- **RO format** (Register to indexed Operand operations)

RO	OpCode	RR or R	R	offset	RESERVED
0 1	2 7	8 13	14 19	20 24	25 31

- **RI** format (Immediate value into Register Loading)

RI	value/address
0 1	2 31

We may dicard this format in the future. If we want move a large number(32-bit) in to register, we may do it 16-bits each time using (PUTHI,PUTLO) or using shift operation.

2.4 Arithmetic Operators

Arithmetic operators are allowed to operate on each compatible basic type. It is up to the compiler to gurantee that an operator is applied to correct operands. On the other side it is up to the architecture to select the appropriate behavior of a certain operator, since the type of the operands is known.

2.4.1 Integer Arithmetic Operators

ADD – Add two operands

Usage:

ADD RS1, RS2, RD

R	ADD___0	RS1	RS2	RD	RES.

Description:

Performs addition and store result in destination.

Operation:

RD ← (RS1 + RS2)

SUB – Subtract two operands

Usage:

SUB RS1, RS2, RD

R	SUB___0	RS1	RS2	RD	RES.

Description:

Performs subtraction and store result in destination.

Operation:

RD ← (RS1 − RS2)

MUL – Multiply two operands

Usage:

MUL RS1, RS2, RD

R	MUL___0	RS1	RS2	RD	RES.

Description:

Performs multiplication and store result in destination.

Operation:

RD ← (RS1 × RS2)

DIV – Divide two operands

Usage:

DIV RS1, RS2, RD

R	DIV___0	RS1	RS2	RD	RES.

Description:

Performs division and store result in destination.

Operation:

RD ← (RS1 / RS2)

MOD – Modulo of two operands

Usage:

MOD RS1, RS2, RD

R	MOD___0	RS1	RS2	RD	RES.

Description:

Performs modulo and store result in destination.

Operation:

RD ← mod(RS1, RS2)

NEG – Change sign to integer operand

Usage:

NEG RS, RD

R	NEG___0	RS	RD	0	RES.

Description:

Performs sign change and store result in destination.

Operation:

RD ← -(RS)

MAX – Maximum between two operands

Usage:

MAX RS1, RS2, RD

R	MAX___0	RS1	RS2	RD	RES.

Description:

Calcualte maximum and store result in destination.

Operation:

RD ← max(RS1, RS2)

MIN – Minimum between two operands

Usage:

MIN RS1, RS2, RD

R	MIN___0	RS1	RS2	RD	RES.

Description:

Calcualte minimum and store result in destination.

Operation:

RD ← min(RS1, RS2)

ABS – Absolute value

Usage:

ABS RS, RD

R	ABS___0	RS	RD	0	RES.

Description:

Calculate absolute value and store result in destination.

Operation:

RD ← |RS|

SHL – Bitwiae Shift Left

Usage:

SHL RS1, RS2, RD

R	SHL___0	RS1	RS2	RD	RES.

Description:

Performs bitwise shift left and store result in destination.

Operation:

RD ← (RS1 << RS2)

SHR – Bitwise Shift Right

Usage:

SHR RS1, RS2, RD

R	SHR___0	RS1	RS2	RD	RES.

Description:

Performs bitwise shift right and store result in destination.

Operation:

RD ← (RS1 >> RS2)

BAND – Bitwise AND of two operands

Usage:

BAND RS1, RS2, RD

R	BAND___0	RS1	RS2	RD	RES.

Description:

Perform bitwise AND and store result in destination.

Operation:

RD ← (RS1 & RS2)

XOR – Bitwise XOR of two operands

Usage:

XOR RS1, RS2, RD

R	XOR___0	RS1	RS2	RD	RES.

Description:

Perform bitwise XOR and store result in destination.

Operation:

RD ← (RS1 ∧ RS2)

BOR – Bitwise OR of two operands

Usage:

BOR RS1, RS2, RD

R	BOR_0	RS1	RS2	RD	RES.

Description:

Perform bitwise OR and store result in destination.

Operation:

RD ← (RS1 | RS2)

BNOT – Bitwise NOT of one operands

Usage:

BNOT RS, RD

R	BNOT_0	RS	RD	0	RES.

Description:

Perform bitwise NOT and store result in destination.

Operation:

RD ← ∼(RS)

AND – Logical AND of two operands

Usage:

AND RS1, RS2, RD

R	AND_0	RS1	RS2	RD	RES.

Description:

Performs logical AND and store result in destination.

Operation:

RD ← (RS1 ∩ RS2)

OR – Logical OR of two operands

Usage:

OR RS1, RS2, RD

R	OR_0	RS1	RS2	RD	RES.

Description:

Performs logical OR and store result in destination.

Operation:

RD ← (RS1 ∪ RS2)

NOT – Logical NOT

Usage:

NOT RS, RD

R	NOT_0	RS	RD	0	RES.

Description:

Performs logical NOT and store result in destination.

Operation:

RD ← not(RS)

2.4.2 Floating Arithmetic Operators

FADD – Add two floating point operands

Usage:

FADD FRS1, FRS2, FRD

R	FADD_0	FRS1	FRS2	FRD	RES.

Description:

Performs floating point addition and store result in destination.

Operation:

FRD ← (FRS1 + FRS2)

FSUB – Subtract two floating point operands

Usage:

FSUB FRS1, FRS2, FRD

R	FSUB__0	FRS1	FRS2	FRD	RES.

Description:

Performs floating pointsubtraction and store result in destination.

Operation:

FRD ← (FRS1 − FRS2)

FMUL – Multiply two floating point operands

Usage:

FMUL FRS1, FRS2, FRD

R	FMUL__0	FRS1	FRS2	FRD	RES.

Description:

Performs floating point multiplication and store result in destination.

Operation:

FRD ← (FRS1 × FRS2)

FDIV – Divide two floating point operands

Usage:

FDIV FRS1, FRS2, FRD

R	FDIV__0	FRS1	FRS2	FRD	RES.

Description:

Performs floating point division and store result in destination.

Operation:

FRD ← (FRS1 / FRS2)

FLR – Floor value

Usage:

FLR FRS, FRD

R	FLR__0	FRS	FRD	0	RES.

Description:

Calculate floor value and store result in destination.

Operation:

FRD ← $\lfloor$FRS$\rfloor$

CEIL – Ceiling value

Usage:

CEIL FRS, FRD

R	CEIL__0	FRS	FRD	0	RES.

Description:

Calculate ceiling value and store result in destination.

Operation:

FRD ← $\lceil$FRS$\rceil$

FABS – Absolute value

Usage:

FABS FRS, FRD

R	FABS__0	FRS	FRD	0	RES.

Description:

Calculate floating point absolute value and store result in destination.

Operation:

FRD ← |FRS|

FNEG – Change sign to floating point operand

Usage:

FNEG FRS, FRD

R	FNEG__0	FRS	FRD	0	RES.

Description:

Performs sign change and store result in destination.

Operation:

FRD ← -(FRS)

2.5 Compare Operators

LT – Less Than

Usage:

LT RS1, RS2, RD

R	LT___0	RS1	RS2	RD	RES.

Description:

Performs integer comparison and store result in destination.

Operation:

RD ← (RS1 $<$ RS2)

LE – Less than or Equal

Usage:

LE RS1, RS2, RD

R	LE___0	RS1	RS2	RD	RES.

Description:

Performs integer comparison and store result in destination.

Operation:

RD ← (RS1 $\leq$ RS2)

EQ – EQual to

Usage:

EQ RS1, RS2, RD

R	EQ___0	RS1	RS2	RD	RES.

Description:

Performs integer comparison and store result in destination.

Operation:

RD ← (RS1 $==$ RS2)

NE – Not Equal to

Usage:

NE RS1, RS2, RD

R	NE___0	RS1	RS2	RD	RES.

Description:

Performs integer comparison and store result in destination.

Operation:

RD ← (RS1 $\neq$ RS2)

GE – Greater than or Equal to

Usage:

GE RS1, RS2, RD

R	GE___0	RS1	RS2	RD	RES.

Description:

Performs integer comparison and store result in destination.

Operation:

RD ← (RS1 $\geq$ RS2)

GT – Greater Than

Usage:

GT RS1, RS2, RD

R	GT___0	RS1	RS2	RD	RES.

Description:

Performs integer comparison and store result in destination.

Operation:

RD ← (RS1 $>$ RS2)

FLT – Less Than

Usage:

FLT FRS1, FRS2, FRD

R	FLT___0	FRS1	FRS2	FRD	RES.

Description:

Performs floating point comparison and store result in destination.

Operation:

FRD $\leftarrow$ (FRS1 $<$ FRS2)

FLE – Less than or Equal

Usage:

FLE FRS1, FRS2, FRD

R	FLE___0	FRS1	FRS2	FRD	RES.

Description:

Performs floating point comparison and store result in destination.

Operation:

FRD $\leftarrow$ (FRS1 $\leq$ FRS2)

FEQ – EQual to

Usage:

FEQ FRS1, FRS2, FRD

R	FEQ___0	FRS1	FRS2	FRD	RES.

Description:

Performs floating point comparison and store result in destination.

Operation:

FRD $\leftarrow$ (FRS1 $==$ FRS2)

FNE – Not Equal to

Usage:

FNE FRS1, FRS2, FRD

R	FNE___0	FRS1	FRS2	FRD	RES.

Description:

Performs floating point comparison and store result in destination.

Operation:

FRD $\leftarrow$ (FRS1 $\neq$ FRS2)

FGE – Greater than or Equal to

Usage:

FGE FRS1, FRS2, FRD

R	FGE___0	FRS1	FRS2	FRD	RES.

Description:

Performs floating point comparison and store result in destination.

Operation:

FRD $\leftarrow$ (FRS1 $\geq$ FRS2)

FGT – Greater Than

Usage:

FGT FRS1, FRS2, FRD

R	FGT___0	FRS1	FRS2	FRD	RES.

Description:

Performs floating point comparison and store result in destination.

Operation:

FRD $\leftarrow$ (FRS1 $>$ FRS2)

2.6 Global Register Set Arithmetic Operators

GADD – Add two global operands

Usage:

GADD GRS1, GRS2, GRD

R	GADD__0	GRS1	GRS2	GRD	RES.

Description:

Performs integer addition and store result in destination.

Operation:

GRD ← (GRS1 + GRS2)

GSUB – Subtract two global operands

Usage:

GSUB GRS1, GRS2, GRD

R	GSUB__0	GRS1	GRS2	GRD	RES.

Description:

Performs integer subtraction and store result in destination.

Operation:

GRD ← (GRS1 − GRS2)

GMUL – Multiply two global operands

Usage:

GMUL GRS1, GRS2, GRD

R	GMUL__0	GRS1	GRS2	GRD	RES.

Description:

Performs integer multiplication and store result in destination.

Operation:

GRD ← (GRS1 × GRS2)

Note:

Adding global register set is according to the compiler writer request, we do not promote this idea.

2.7 Type Conversion Operators

Type conversion operaters are needed to modify the type of the content of a register before applying a certain arithmetic operation, in order to perform the correct arithmetic function.

TBL – Convert To Boolean Type

Usage:

TBL RS, RD

R	TBL__0	RS	RD	0	RES.

Description:

Performs conversion and store result in destination.

Operation:

RD ← bool(RS)

TCH – Convert To Character Type

Usage:

TCH RS, RD

R	TCH__0	RS	RD	0	RES.

Description:

Performs conversion and store result in destination.

Operation:

RD ← char(RS)

TRL – Convert To Real Type

Usage:

TRL RS, FRD

R	TRL__0	RS	FRD	0	RES.

Description:

Performs conversion and store result in destination.

Operation:

FRD ← real(RS)

TDB – Convert To Double Type

Usage:

TDB RS, DRD

R	TDB__0	RS	DRD	0	RES.

Description:

Performs conversion and store result in destination.

Operation:

DRD ← double(RS)

TIN – Convert To Integer Type

Usage:

TIN FRS, RD

TIN DRS, RD

R	TIN__0	FRS	RD	0	RES.
R	TIN__0	DRS	RD	0	RES.

Description:

Performs conversion and store result in destination.

Operation:

RD ← int(FRS or DRS)

2.8 Data Movement

MOVE – Move data between interger registers

Usage:

MOVE RS, RD

R	MOVE__0	RS	RD	0	RES.

Description:

Perform move and copy source to destination

Operation:

RD ← (RS)

FMOVE – Move data between floating registers

Usage:

FMOVE FRS, FRD

R	FMOVE__0	FRS	FRD	0	RES.

Description:

Perform move and copy source to destination

Operation:

FRD ← FRS

GTL – Move data from global register to local register

Usage:

GTL GRS, RD

R	GTL__0	GRS	RD	0	RES.

Description:

Perform move from global to local

Operation:

RD ← (GRS)

LTG – Move data from local register to global register

Usage:

LTG RS, GRD

R	LTG__0	RS	GRD	0	RES.

Description:

Perform move from local to global

Operation:

GRD ← (RS)

PUTR1 – Put immediate data into register R1

Usage:

PUTR1 value	Put immediate value/address into R1	RI	PUTR1	value		

Description:

Put immediate value/address into R1

Operation:

R1 ← value

Note:

This instruction is not meaningful. If we want to load very large integer number, we can follow MIPS convetion LOAD lower half and load upper half.

PUTR – Load sign-extended immediate data into register RD

Usage:

PUTR value, RD	Put sign-extended immediate value into RD	RI	PUTR	RD	value	

Description:

Put immediate value/address into RD

Operation:

RD ← value

LOAD – Load data from Frame

Usage:

LOAD RS1 | RS2, RD	Loads data from ≪RS1, RS2≫ into RD	R	LOAD_0	RS1	RS2	RD	RES.	+
LOAD RS | offst, RD	Loads data from ≪RS, offst≫ into RD	RO	LOAD_0	RD	RS	offst	RES.	*

Description:

Loads frame-data into register(s)

Operation:

RD ← ≪RS1, RS2≫ (or F ← RS; RD ← F[offst])

Note:

The maximum value of 'offst' is 31 ($2^5 - 1$). The instruction has no effect if the data is not present (i.e. it's non-blocking).

STORE – Store data into Frame

Usage:

STORE RS, RD1 | RD2	Stores data from RS into ≪RD1, RD2≫	R	STORE_0	RS	RD1	RD2	RES.	+
STORE RS, RD | offst	Stores data from RS into ≪RD, offst≫	RO	STORE_0	RS	RD	offst	RES.	*

Description:

Stores register value into single frame-destination.

Operation:

≪RD1, RD2≫ ← RS (or F ← RD; F[offst] ← RS or F ← RD)

Note:

The maximum value of 'offst' is 31 ($2^5 - 1$). The instruction has no effect if the data is not present (i.e. it's non-blocking).

2.9 I-Structure Management

IALLOC – Allocate memory for an I-Structure

Usage:

IALLOC RS, RD	Allocates an I-Structure of RS entries	R	IALLOC0	RS	RD	0	RES.	+
IALLOC value, RD	Allocates an I-Structure of 'value' entries	trans: **PUTR1 value; IALLOC R1, RD**						

Description:

An I-Structure of the specified size is allocated. The I-Structure pointer is stored in RD.

Operation:

RD ← &I

I-Structure flags are initialized to E (Empty)

IFREE – Free the memory belonging to a given I-Structure

Usage:

IFREE RS	Frees the specified I-Structure	R	IFREE_0	RS	0	0	RES.	*
IFREE addr	Frees the specified I-Structure	trans:	**PUTR1 addr; IFREE R1**					

Description:

The I-Structure specified by RS is freed.

Operation:

-

IFETCH – Fetch an I-Structure entry

Usage:

IFETCH RS1, RS2, RD	Fetches <RS1, RS2>	R	IFETCH0	RS1	RS2	RD	RES.	+
IFETCH RS | index, RD	Fetches <RS, index>	RO	IFETCH0	RD	RS	index	RES.	*

Description:

Given the I-Structure I, it loads the specified value into RD if <I, index>.flag is F (data present), else the request is queued, and the flag is set to W (Waiting for data to come).

Operation:

RD ← I[index].value   IF I[index].flag == F

ISTORE – Store an I-Structure entry

Usage:

ISTORE RS, RD1 | RD2	Stores into <RD1, RD2>	R	ISTORE0	RS	RD1	RD2	RES.	+
ISTORE RS, RD | index	Stores into <RD, index>	RO	ISTORE0	RS	RD	index	RES.	

Description:

Given the I-Structure I, it stores the value specified in RD and set <I, index>.flag to F (data present).

Operation:

I[index].value ← RS and I[index].flag ← F (thereafter, all pending requests are satisfied).

READ – Fetch a memory entry

Usage:

READ RS1, RS2, RD	Fetches <RS1, RS2>	R	READ0	RS1	RS2	RD	RES.	+
READ RS | index, RD	Fetches <RS, index>	RO	READ0	RD	RS	index	RES.	*

Description:

READ general memory.

Operation:

WRITE – Store a memory entry

Usage:

WRITE RS, RD1 | RD2	Stores into <RD1, RD2>	R	WRITE0	RS	RD1	RD2	RES.	+
WRITE RS, RD | index	Stores into <RD, index>	RO	ISTORE0	RS	RD	index	RES.	

Description:

Write general memory.

Operation:

2.10 Thread Support

FORKSP – Schedule the execution of code on Synchronization Processor

Usage:

FORKSP RS, RD	conditionally schedules the code at RD	R	FORKSP0	RS	RD		RES.	+
FORKSP RD	unconditionally schedules the code at RD	R	FORKSP0	0	RD		RES.	+
FORKSP RS, addr	conditionally schedules the code at addr	trans:	**PUTR1 addr; FORKSP RS, R1**					
FORKSP addr	unconditionally schedules the code at addr	trans:	**PUTR1 addr; FORKSP R1**					

Description:

Schedule the execution of a certain thread on SP. When present, the condition is true if its value is not zero.

Operation:

-

FORKEP – Schedule the execution of code on Execution Processor

Usage:

FORKEP RS, RD	conditionally schedules the code at RD	R	FORKEP0	RS	RD		RES.	+
FORKEP RD	unconditionally schedules the code at RD	R	FORKEP0	0	RD		RES.	+
FORKEP RS, addr	conditionally schedules the code at addr	trans:	**PUTR1 addr; FORKEP RS, R1**					
FORKEP addr	unconditionally schedules the code at addr	trans:	**PUTR1 addr; FORKEP R1**					

Description:

Schedule the execution of a certain thread on EP. When present, the condition is true if its value is not zero.

Operation:

-

STOP – Terminate the current thread

Usage:

STOP	R	STOP␣0	0	0	0	RES.	

Description:

Stop the current thread and schedule another one. This also frees the Running Context.

Operation:

-

2.11 I/O Instructions

INPUT – Input data from a device

Usage:

INPUT index, RD	Inputs data from device number 'index'	RO	INPUT␣0	RD	0	index	RES.	*

Description:

Inputs data from the given device into destination

Operation:

RD ← D[index]

OUTPUT – Output data to a device

Usage:

OUTPUT RS, index	Outputs data to device number 'index'	RO	OUTPUT0	RS	0	index	RES.	*

Description:

Outputs data to the given device

Operation:

D[index] ← RS

FINPUT – Input floating point data from a device

Usage:

FINPUT index, FRD	Inputs floating point data from device number 'index'	RO	FINPUT␣0	FRD	0	index	RES.	*

Description:

Inputs data from the given device into destination

Operation:

FRD ← D[index]

FOUTPUT – Output floating data to a device

Usage:

FOUTPUT FRS, index	Outputs floating data to device number 'index'	RO	FOUTPUT0	FRS	0	index	RES.	*

Description:

Outputs data to the given device

Operation:

D[index] ← FRS

2.12 System Calls

System calls are needed to invoke those opearations that are cannot be implemented directly at architectural level. The architecture may provide support for the implementation of system calls.

SC – Launch the specified System Call

Usage:

SC #sc_id RS	RO	SC___0	RS	0	sc_id	RES.	+
SC #sc_id RRS	RO	SC___1	RRS	0	sc_id	RES.	+
SC #sc_id RS, RD	RO	SC___0	RS	RD	sc_id	RES.	+
SC #sc_id RRS, RD	RO	SC___1	RRS	RD	sc_id	RES.	+
SC #sc_id addr, RD	trans: **PUTR1 addr; SC #sc_id R1,RS**						

Description:

Invoke the System Call ’sc_id’ with RS (RRS) as input parameters and, eventually RD as output parameters

Operation:

2.12.1 Frame Management

SC #FALLOC – System Call: Associate a frame to a code-block

Usage:

SC #FALLOC RRS, RD	Returns in RD the address of the frame	RO	SC___0	RRS	RD	FALLOC	RES.	*

Description:

A frame F is allocated and its address stored in RD for the code-block whose address has been specified in RS1 and whose synchronization count is specifed in RS2.

Operation:

EP: allocates a frame; requests SP to run the frame initialization routine; RD ← &F.

SP: executes the frame initialization routine.

Note:

There’s no stall in EP.

SC #FFREE – System Call: Free the frame associated with current code-block

Usage:

SC #FFREE RS	Free the frame pointed by RS	RO	SC___0	RS	0	FFREE	RES.	*

Description:

-

Operation:

-

SC #FREALLOC– System Call: Set the value of current code-block synchronization-count

Usage:

SC #FRALLOC RS	RO	SC___0	RS	0	FFREE	RES.	*

Description:

Set the value of current code-block synchronization-count to what specified in RS

Operation:

Sync-Count of F ← RS

3 Pragmas

The pragmas are directives to the compiler that are useful to identify features of the code.

- **VERSION string** specifies the version number of current program
- **CODE string** specifies the name of the code block
- **THREAD string** specifies the beginning of a thread
- **END** specifies the end of a code block

4 Frame Usage Conventions

A Frame is a (local) chunk of memory, which holds all the data which are addressed by a certain code-block. The following conventions apply to the a frame.

> ...

5 Thread Management Conventions

(to be written) ...

6 Possible Instruction Set Extensions

From IF1 graph analysis, it appears that could be usuful to introduce:

- Support for Trascendental Operators
- Support for Reduce Operators
- Support for Vector Operators
- Support for Double operand type (sign, 52-bit mantissa, 11-bit exponent(64-bit double-precision IEEE754))

Appendix A – Compatibility with previous notations

Register notation

- R0, R2, . . . were previously used to indicate RR0, RR2, . . .

Frame Management

- **MKTAG RD, RS, offst** instruction is not necessary any more, since:
  LOAD RS |offst, RD
  prepares automatically the pointer to frame entry $\ll$RS, offst$\gg$.

- **FALLOC addr, RD** instruction is translated into:
  PUTR1 addr
  SC #FALLOC R1, RD
  This has the advantage of allowing to specify any possible address within memory.

- **FFREE RS** instruction is transalted into:
  SC #FFREE RS

- **FREALLOC value** instruction is transalted into:
  PUTR1 value
  SC #FRALLOC R1
  This has the advantage of allowing to specify any possible value between 0 and 2^{30}.

Data Movement

- **STOREI value, RS, offst** instruction is transalted into:
  PUTR1 value
  STORE R1, RS, offst
  This has the advantage of allowing to specify any possible value between 0 and 2^{30}.

- **LAOD2 RS |offst, RRD** (or **LAOD RS |offst, RD1 |RD2**) instruction has been removed.

Thread Support

- **BR**, **FORK.P** and **SWITCH.P** instructions are replaced by **FORKEP**

- **FORK.S**, and **SWITCH.S** instruction are replaced by **FORKSP**

Pragmas

- **SYNC** pragmas is not needed since the synchronization count is specified when FALLOC system call is invoked.

Appendix B – List of Op-Codes

OpCode	R format	RO format	RI format
ϕ			X
FORKSP0	X		
FORKEP0	X		
IALLOC0	X		
IFREE_0	X		
IFETCH0	X	X	
ISTORE0	X	X	
READ0	X	X	
WRITE0	X	X	
LOAD__0	X	X	
STORE_0	X	X	
STORE_1	X	X	
MOVE__0	X		
STOP__0	X		
ADD___0	X		
SUB___0	X		
MUL___0	X		
DIV___0	X		
MOD___0	X		
AND___0	X		
OR____0	X		
NOT___0	X		
SHL___0	X		
SHR___0	X		
BOR___0	X		
BNOT__0	X		
BAND__0	X		
BXOR__0	X		
NEG___0	X		
MAX___0	X		
MIN___0	X		
ABS___0	X		
EXP___0	X		
LT____0	X		
LE____0	X		
EQ____0	X		
NE____0	X		
GE____0	X		
GT____0	X		
FLT___0	X		
FLE___0	X		
FEQ___0	X		
FNE___0	X		
FGE___0	X		
FGT___0	X		
FADD__0	X		
FSUB__0	X		
FMUL__0	X		
FDIV__0	X		
CEIL__0	x		
FLR___0	X		
FABS__0	X		
GADD__0	X		
GSUB__0	X		
GMUL__0	X		
GTL___0	X		
LTG___0	X		
TBL___0	X		
TCH___0	X		
TRL___0	X		
TDB___0	X		
TIN___0	X		
SC____0		X	
SC____1		X	
INPUT_0		X	
OUTPUT0		X	
BOR___0	X		
68	36	9	1

The TLS (thread levelspeculation) instructions are not added yet.