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 implementation 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 thread 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 implementation 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 be represented by READ and WRITE instructions. I-structure semantics 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 can be used to improve access time.

Global memory, regardless of the semantics being applied will be accessed 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 thread 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 implementation, 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 accommodate 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.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Character</td>
<td>0..255</td>
</tr>
<tr>
<td>Integer</td>
<td>-2147483648 .. 2147483647 (−2^{31} .. 2^{31} − 1)</td>
</tr>
<tr>
<td>Real</td>
<td>32-bit single-precision or 64-bit double-precision (IEEE 754 standard)</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>R</th>
<th>OpCode</th>
<th>RS1</th>
<th>RS2</th>
<th>RD</th>
<th>RESERVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>19</td>
<td>20</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>25</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>RO</th>
<th>OpCode</th>
<th>RR or R</th>
<th>R</th>
<th>offset</th>
<th>RESERVED</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>19</td>
<td>20</td>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>25</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>RI</th>
<th>value/address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

We may discard 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 guarantee 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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Usage</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADD</strong></td>
<td>Add two operands</td>
<td>ADD RS1, RS2, RD</td>
<td>Performs addition and store result in destination.</td>
<td>RD ← (RS1 + RS2)</td>
</tr>
<tr>
<td><strong>SUB</strong></td>
<td>Subtract two operands</td>
<td>SUB RS1, RS2, RD</td>
<td>Performs subtraction and store result in destination.</td>
<td>RD ← (RS1 − RS2)</td>
</tr>
<tr>
<td><strong>MUL</strong></td>
<td>Multiply two operands</td>
<td>MUL RS1, RS2, RD</td>
<td>Performs multiplication and store result in destination.</td>
<td>RD ← (RS1 × RS2)</td>
</tr>
<tr>
<td><strong>DIV</strong></td>
<td>Divide two operands</td>
<td>DIV RS1, RS2, RD</td>
<td>Performs division and store result in destination.</td>
<td>RD ← (RS1 / RS2)</td>
</tr>
<tr>
<td><strong>MOD</strong></td>
<td>Modulo of two operands</td>
<td>MOD RS1, RS2, RD</td>
<td>Performs modulo and store result in destination.</td>
<td>RD ← mod(RS1, RS2)</td>
</tr>
<tr>
<td><strong>NEG</strong></td>
<td>Change sign to integer operand</td>
<td>NEG RS, RD</td>
<td>Performs sign change and store result in destination.</td>
<td></td>
</tr>
</tbody>
</table>
**MAX** – Maximum between two operands

**Usage:**

\[ \text{MAX RS1, RS2, RD} \]

**Description:**
Calcualte maximum and store result in destination.

**Operation:**
\[ \text{RD} \leftarrow \max(\text{RS1}, \text{RS2}) \]

**MIN** – Minimum between two operands

**Usage:**

\[ \text{MIN RS1, RS2, RD} \]

**Description:**
Calcualte minimum and store result in destination.

**Operation:**
\[ \text{RD} \leftarrow \min(\text{RS1}, \text{RS2}) \]

**ABS** – Absolute value

**Usage:**

\[ \text{ABS RS, RD} \]

**Description:**
Calculate absolute value and store result in destination.

**Operation:**
\[ \text{RD} \leftarrow |\text{RS}| \]

**SHL** – Bitwise Shift Left

**Usage:**

\[ \text{SHL RS1, RS2, RD} \]

**Description:**
Performs bitwise shift left and store result in destination.

**Operation:**
\[ \text{RD} \leftarrow (\text{RS1} \ll \text{RS2}) \]

**SHR** – Bitwise Shift Right

**Usage:**

\[ \text{SHR RS1, RS2, RD} \]

**Description:**
Performs bitwise shift right and store result in destination.

**Operation:**
\[ \text{RD} \leftarrow (\text{RS1} \gg \text{RS2}) \]

**BAND** – Bitwise AND of two operands

**Usage:**

\[ \text{BAND RS1, RS2, RD} \]

**Description:**
Perform bitwise AND and store result in destination.

**Operation:**
\[ \text{RD} \leftarrow (\text{RS1} \& \text{RS2}) \]

**XOR** – Bitwise XOR of two operands

**Usage:**

\[ \text{XOR RS1, RS2, RD} \]

**Description:**
Perform bitwise XOR and store result in destination.
Perform bitwise XOR and store result in destination.

**Operation:**
RD ← (RS1 \( \land \) RS2)

### BOR – Bitwise OR of two operands

**Usage:**
BOR RS1, RS2, RD

**Description:**
Perform bitwise OR and store result in destination.

**Operation:**
RD ← (RS1 \( \lor \) RS2)

### BNOT – Bitwise NOT of one operand

**Usage:**
BNOT RS, RD

**Description:**
Perform bitwise NOT and store result in destination.

**Operation:**
RD ← \(~\)RS

### AND – Logical AND of two operands

**Usage:**
AND RS1, RS2, RD

**Description:**
Performs logical AND and store result in destination.

**Operation:**
RD ← (RS1 \( \cap \) RS2)

### OR – Logical OR of two operands

**Usage:**
OR RS1, RS2, RD

**Description:**
Performs logical OR and store result in destination.

**Operation:**
RD ← (RS1 \( \cup \) RS2)

### NOT – Logical NOT

**Usage:**
NOT RS, RD

**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

**Description:**
Performs floating point addition and store result in destination.

**Operation:**
FSUB – Subtract two floating point operands

Usage:
FSUB FRS1, FRS2, FRD

Description:
Performs floating point subtraction and store result in destination.

Operation:
FRD ← (FRS1 – FRS2)

FMUL – Multiply two floating point operands

Usage:
FMUL FRS1, FRS2, FRD

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

Description:
Performs floating point division and store result in destination.

Operation:
FRD ← (FRS1 / FRS2)

FLR – Floor value

Usage:
FLR FRS, FRD

Description:
Calculate floor value and store result in destination.

Operation:
FRD ← ⌊FRS⌋

CEIL – Ceiling value

Usage:
CEIL FRS, FRD

Description:
Calculate ceiling value and store result in destination.

Operation:
FRD ← ⌈FRS⌉

FABS – Absolute value

Usage:
FABS FRS, FRD

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

Description:
Performs sign change and store result in destination.
2.5 Compare Operators

**LT** – Less Than

**Usage:**

\[ LT \, RS1, \, RS2, \, RD \]

**Description:**
Performs integer comparison and store result in destination.

**Operation:**
\[ RD \leftarrow (RS1 < RS2) \]

**LE** – Less than or Equal

**Usage:**

\[ LE \, RS1, \, RS2, \, RD \]

**Description:**
Performs integer comparison and store result in destination.

**Operation:**
\[ RD \leftarrow (RS1 \leq RS2) \]

**EQ** – Equal to

**Usage:**

\[ EQ \, RS1, \, RS2, \, RD \]

**Description:**
Performs integer comparison and store result in destination.

**Operation:**
\[ RD \leftarrow (RS1 == RS2) \]

**NE** – Not Equal to

**Usage:**

\[ NE \, RS1, \, RS2, \, RD \]

**Description:**
Performs integer comparison and store result in destination.

**Operation:**
\[ RD \leftarrow (RS1 \neq RS2) \]

**GE** – Greater than or Equal to

**Usage:**

\[ GE \, RS1, \, RS2, \, RD \]

**Description:**
Performs integer comparison and store result in destination.

**Operation:**
\[ RD \leftarrow (RS1 \geq RS2) \]

**GT** – Greater Than

**Usage:**

\[ GT \, RS1, \, RS2, \, RD \]

**Description:**
Performs integer comparison and store result in destination.

**Operation:**
\[ RD \leftarrow (RS1 > RS2) \]
**FLT** – Less Than

**Usage:**

\[
\text{FLT FRS1, FRS2, FRD}
\]

**Description:**

Performs floating point comparison and store result in destination.

**Operation:**

\[
\text{FRD} \leftarrow (\text{FRS1} < \text{FRS2})
\]

**FLE** – Less than or Equal

**Usage:**

\[
\text{FLE FRS1, FRS2, FRD}
\]

**Description:**

Performs floating point comparison and store result in destination.

**Operation:**

\[
\text{FRD} \leftarrow (\text{FRS1} \leq \text{FRS2})
\]

**FEQ** – EQual to

**Usage:**

\[
\text{FEQ FRS1, FRS2, FRD}
\]

**Description:**

Performs floating point comparison and store result in destination.

**Operation:**

\[
\text{FRD} \leftarrow (\text{FRS1} == \text{FRS2})
\]

**FNE** – Not Equal to

**Usage:**

\[
\text{FNE FRS1, FRS2, FRD}
\]

**Description:**

Performs floating point comparison and store result in destination.

**Operation:**

\[
\text{FRD} \leftarrow (\text{FRS1} \neq \text{FRS2})
\]

**FGE** – Greater than or Equal to

**Usage:**

\[
\text{FGE FRS1, FRS2, FRD}
\]

**Description:**

Performs floating point comparison and store result in destination.

**Operation:**

\[
\text{FRD} \leftarrow (\text{FRS1} \geq \text{FRS2})
\]

**FGT** – Greater Than

**Usage:**

\[
\text{FGT FRS1, FRS2, FRD}
\]

**Description:**

Performs floating point comparison and store result in destination.

**Operation:**

\[
\text{FRD} \leftarrow (\text{FRS1} > \text{FRS2})
\]

### 2.6 Global Register Set Arithmetic Operators

**GADD** – Add two global operands

**Usage:**

\[
\text{GADD}
\]
GADD GRS1, GRS2, GRD

Description:
Performs integer addition and store result in destination.

Operation:
GRD ← (GRS1 + GRS2)

GSUB – Subtract two global operands

Usage:
GSUB GRS1, GRS2, GRD

Description:
Performs integer subtraction and store result in destination.

Operation:
GRD ← (GRS1 − GRS2)

GMUL – Multiply two global operands

Usage:
GMUL GRS1, GRS2, GRD

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 operators 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

Description:
Performs conversion and store result in destination.

Operation:
RD ← bool(RS)

TCH – Convert To Character Type

Usage:
TCH RS, RD

Description:
Performs conversion and store result in destination.

Operation:
RD ← char(RS)

TRL – Convert To Real Type

Usage:
TRL RS, FRD

Description:
Performs conversion and store result in destination.

Operation:
FRD ← real(RS)

**TDB**  – Convert To Double Type

**Usage:**
TDB RS, DRD

**Description:**
Performs conversion and store result in destination.

**Operation:**
DRD ← double(RS)

**Usage:**
TIN FRS, RD
TIN DRS, RD

**Description:**
Performs conversion and store result in destination.

**Operation:**
RD ← int(FRS or DRS)

---

### 2.8 Data Movement

**MOVE**  – Move data between integer registers

**Usage:**
MOVE RS, RD

**Description:**
Perform move and copy source to destination

**Operation:**
RD ← (RS)

**FMOVE**  – Move data between floating registers

**Usage:**
FMOVE FRS, FRD

**Description:**
Perform move and copy source to destination

**Operation:**
FRD ← FRS

**GTL**  – Move data from global register to local register

**Usage:**
GTL GRS, RD

**Description:**
Perform move from global to local

**Operation:**
RD ← (GRS)

**LTG**  – Move data from local register to global register

**Usage:**
LTG RS, GRD

**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

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 convention 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

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

LOAD RS | offset, RD
Loads data from RS, offset into RD

Description:
Loads frame-data into register(s)

Operation:

Note:
The maximum value of ’offset’ 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

STORE RS, RD | offset
Stores data from RS into RD, offset

Description:
Stores register value into single frame-destination.

Operation:

Note:
The maximum value of ’offset’ 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

IALLOC value, RD
Allocates an I-Structure of ’value’ entries

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
IFREE addr Frees the specified I-Structure

description:
The I-Structure specified by RS is freed.

Operation:
-

IFETCH – Fetch an I-Structure entry

Usage:
IFETCH RS1, RS2, RD Fetches <RS1, RS2>
IFETCH RS | index, RD Fetches <RS, index>

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>
ISTORE RS, RD | index Stores into <RD, index>

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>
READ RS | index, RD Fetches <RS, index>

Description:
READ general memory.

Operation:

WRITE – Store a memory entry

Usage:
WRITE RS, RD1 | RD2 Stores into <RD1, RD2>
WRITE RS, RD | index Stores into <RD, index>

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
FORKSP RD unconditionally schedules the code at RD
FORKSP RS, addr conditionally schedules the code at addr
FORKSP addr unconditionally schedules the code at addr

Description:
Schedule the execution of a certain thread on SP. When present, the condition is true if its value is not zero.
FORKEP — Schedule the execution of code on Execution Processor

Usage:
- FORKEP RS, RD conditionally schedules the code at RD
- FORKEP RD unconditionally schedules the code at RD
- FORKEP RS, addr conditionally schedules the code at addr
- FORKEP addr unconditionally schedules the code at addr

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

STOP — Terminate the current thread

Usage:
STOP

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

2.11 I/O Instructions

INPUT — Input data from a device

Usage:
- INPUT index, RD inputs data from device number ‘index’

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’

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’

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’

Description:
Outputs data to the given device

Operation:
2.12 System Calls

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

<table>
<thead>
<tr>
<th>SC</th>
<th>Launch the specified System Call</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Usage:</strong></td>
</tr>
<tr>
<td>SC #sc_id RS</td>
<td>RO</td>
</tr>
<tr>
<td>SC #sc_id RRS</td>
<td>RO</td>
</tr>
<tr>
<td>SC #sc_id RS, RD</td>
<td>RO</td>
</tr>
<tr>
<td>SC #sc_id RRS, RD</td>
<td>RO</td>
</tr>
<tr>
<td>SC #sc_id addr, RD</td>
<td>trans: PUTR1 addr; SC #sc_id R1,RS</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>SC #FALLOC</th>
<th>System Call: Associate a frame to a code-block</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usage:</strong></td>
<td>Returns in RD the address of the frame</td>
</tr>
<tr>
<td>SC #FALLOC RRS, RD</td>
<td>RO</td>
</tr>
</tbody>
</table>

**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 specified 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.

<table>
<thead>
<tr>
<th>SC #FFREE</th>
<th>System Call: Free the frame associated with current code-block</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usage:</strong></td>
<td>Free the frame pointed by RS</td>
</tr>
<tr>
<td>SC #FFREE RS</td>
<td>RO</td>
</tr>
</tbody>
</table>

**Description:**

**Operation:**

<table>
<thead>
<tr>
<th>SC #FREALLOC</th>
<th>System Call: Set the value of current code-block synchronization-count</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Usage:</strong></td>
<td>Set the value of current code-block synchronization-count to what specified in RS</td>
</tr>
<tr>
<td>SC #FREALLOC RS</td>
<td>RO</td>
</tr>
</tbody>
</table>

**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 useful 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 ≪RS, offst≫.
- 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 #FREALLOC 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

<table>
<thead>
<tr>
<th>OpCode</th>
<th>R format</th>
<th>RO format</th>
<th>RI format</th>
</tr>
</thead>
<tbody>
<tr>
<td>φ</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORKSP0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORKEP0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IALLOC0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IFREE0</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>IFETCH0</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ISTORE0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>READ0</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>WRITE0</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>LOAD_0</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>STORE_0</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>STORE_1</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MOVE_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOP_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOD_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHR_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOR_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNOT_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAND_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BXOR_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEG_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIN_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABS_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXP_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LT_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LE_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EQ_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NE_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GE_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GT_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLT_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLE_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FEQ_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FNE_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FGE_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FGT_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSUB_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMUL_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDIV_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CEIL_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLR_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FABS_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GADD_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GSUB_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GMUL_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GTL_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GTG_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TBL_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TCH_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRL_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TDB_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIN_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SC_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SC_1</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INPUT_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT_0</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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