SAFE ISA
(version 3.0 with Interrupts per Thread)

Thomas F. Knight, Jr. André DeHon Andrew Sutherland Sumit Ray
Udit Dhawan Albert Kwon

October 4, 2012

Contents

1 Notes:

This is current proposal of record for the SAFE architecture.

1.1 Threads

The architecture supports threads directly in hardware with a thread pointer and associated thread frame. Each thread has access to a dedicated register file implemented in the thread frame.

1.2 Gates

Gates provide the mechanism for changing authority while executing a procedure or closure. Gates are called from within a single thread without changing the register frame between caller and callee. Gates have an associated environment pointer so they can be used for closures.

2 Atoms

All data in the SAFE machine is stored in "Atoms". An atom consists of a data field, a tag field and an atomic group field. The tag field is used for security processing within the TMU. The use of the tag field is not discussed in this document.

The individual components of an atom, $\mathbf{A}$, are referred to via subscript, $\mathbf{A}$, $\mathbf{A}_{tag}$ and $\mathbf{A}_{ag}$. Conceptually the three parts constitute the one register.

2.1 Atomic Groups

The atomic group field in the atom describes the most basic type system for the machine. Each instruction in the ISA defines which atomic groups the instruction is valid for and the resulting atomic group for any result of the instruction.

The processor understands these atomic groups:

- Frame Pointer
- Constant Frame Pointer
- Linear Frame Pointer [(1) now think this could be a LabelModel — see rules; (2) we probably need to find a different term since this doesn't conform to the PL folks notion of "linear" –AMD]
- Instruction Pointer
- Gate Pointer
- Stream Read Pointer
- Stream Write Pointer
- Thread Pointer
- Forwarding Pointer
- Double (Double Float)
- Integer
- Instruction
- Authority (was previously known as Principal)
- Principal (temporarily known as Principal Name to distinguish from Authority)
- Uninitialized
- Error (NAV)[1]
- Tag (for first-class tag case)

[1]Believe these are the same thing. We started using Errors for cases like out-of-bounds pointer and non-present memory locations before there was a formal Breeze NAV proposal on the table. I believe they are all the same spirit, so there is only one Atomic Group type here. In the Breeze NAV, there is a proposal that the payload be a pointer to something that explains the error. That will require additional clarification.

- Empty
- EOS
- FREE

[It might be possible to combine Uninitialized, Empty, EOS, Free by unifying as Out-of-Band and differentiating in the Value field. Error (NAV) will use value field for a pointer, so it would not be part of this unification. –AMD]

2.1.1 Linear Pointers

[Consider moving to LabelModel. See linear LabelModel in rules document. –AMD]

A subset of the atomic groups are linear pointers, linear pointers are needed in certain cases to maintain the security or correctness properties of the machine. In order to maintain a guarantee that only one copy exists, linear pointers may not be copied. Linear pointers may be moved between registers or between registers and memory using the move instructions. The move instruction maintains linearity by atomically transferring the pointer and destroying the old copy. Using linear pointers with the generic offp and or cprm, cpmr (load/store) operations is an error.

Linear Frame Pointer – The linear frame pointer is provided to enable user code to share memory in a controlled way.

Thread Pointer – A thread pointer represents the capability to access the state of a thread. The **TP** needs to be linear to prevent inconsistencies in the thread state.

Input Stream Pointer – The stream pointer must be linear to allow it to be auto incremented by the stream instructions. If there were another pointer, it would not see the increment and end up pointing to a location now used in the stream.

Output Stream Pointer – The stream pointer must be linear to allow it to be auto incremented by the stream instructions. Same property as above. There will be separate input and output stream pointers to the same stream.

2.2 Validation

Before execution the current instruction **CI** and its operands are validated. The check examines the following fields:

- $\mathbf{CI}_{ag}$ – must always be Instruction_{ag}
- **CI** – determines the allowable atomic groups for operands
- $\mathbf{OP}^{n}{}_{ag}$ – operands must be consistent with **CI**

3 Fat Pointers

All data lives in frames. All frames are referenced by fat pointers. The fat pointers include the base and bounds of the frame as well as the current pointed-to location.

3.1 Bounds Checks

Frame bounds checking automatically ensures

- No code can be executed outside the current code frame
- No memory can be accessed outside the fat pointer frame
- No illegal fat pointers can be created

Any violation triggers a thread switch to the Frame Bounds Error handler thread.

4 Processor State

The processor state that is held in hardware is the following:

- Thread Pointer capturing thread state
- Time State
- Transaction State

4.1 Thread Pointer

The thread state shows up as a single thread pointer **TP** register in the processor state. **TP** holds a pointer to the state of the currently running thread.

4.2 Timer State

Timer state consists of three registers that are processor state but not thread state.

QT – Quanta; the scheduling precision or time slice.

CLK – A countdown timer that may be thought of as the low part of a counter whose range is 0 to the value of **QT**.

SLOT – A countdown timer that may be thought as the high part of the counter, which is decremented when **CLK** reaches 0.

TTT – Timing Trap Thread; thread to load when the timer counts down to zero or when a yield occurs.

We assume **QT** is loaded in some manner from the certified boot image and never changes.

4.3 Transaction State

Transaction state consists of:

TID – Start ID of Transaction in Process – non-zero when a transaction has started but not ended or aborted

Implementation-dependent state as required to support the transaction – abstractly a snapshot of the thread state; in practice a copy of dirty state that may need to be rolled back on transaction abort

instruction counter – we may add counter to limit length?

Transactions do not persist across thread switches, so transaction state is processor state not thread state.

5 Timer Behavior

On each cycle:

```
CLK=CLK-1;
if (CLK==0)
  {
   CLK=QT;
   if (SLOT==0)
      Perform Timer Interrupt
         // Thread-Switch to TTT (and TTT<-NaV)
   else
     SLOT=SLOT-1;
  }
else
  {}
```

6 Thread Frame

The frame of memory pointed to by the **TP** register holds the state associated with the currently running thread. The register frame is divided into two sections. The first section holds the private state that is inaccessible to the thread. The second part is the register file for use by the thread.

The private state is protected from access through register and memory operations. The currently running thread may not access the private portion of the state because instruction operands have an offset added by the hardware so that there is no way an instruction can address the private portion of the current state as a register. Since the TP is linear, no one can access it through memory operations while it is installed as the current thread.

6.1 Private State

The private state contains the registers necessary to implement the thread.

Name	Offset from TP_{base}	Description	Update on Faults
ITT^0	0x20	TMU Miss	No
ITT^1	0x21	Stream Empty/Full?	No
ITT^2	0x22	Arithmetic Error	No
ITT^3	0x23	Frame Bounds Error	No
ITT^4	0x24	Atomic Group Error	No
ITT^5	0x25	Frame Alignment Error	No
ITT^6	0x26	Gate Stack Bounds Error	No
ITT^7	0x27	Transaction Error	No
ITT^8	0x28	Mangled Thread Error	No
ITT^9	0x29	Return Mismatch Error	No
PC	0x30	Program Counter	No
A	0x31	Authority	No
TS	0x32	Thread Status	Yes
GS	0x33	Gate Stack pointer	No
CI	0x34	Current Instruction	Yes
OP^1	0x35	OPerand n	Yes
OP^2	0x36	OPerand n	Yes
OP^3	0x37	OPerand n	Yes
MR	0x38	Result of Memory Load	Yes
~~**TR**~~	~~0x39~~	~~Result of TMU check~~	No
NC	0x39	Non-Cacheable TMU Entry	No
fPC	0x3A	faulting Program Counter	Yes
fA	0x3B	faulting Authority	Yes
LR	0x3C	Live Register mask	No
WR	0x3D	Writeable Register mask	No
IC	0x3E	Instruction Count	No
IL	0x3F	Instruction Limit	No

The detailed description of each state register is described below.

6.1.1 Program Counter

PC$_{tag}$ – Program counter tag
PC – Program counter

The program counter is a fat pointer into the currently executing instruction frame.

6.1.2 Authority

A – Authority

A – represents the authority that the processor is currently executing under.
There is no separate tag on **A**. The tag on **A** is $\mathbf{PC}_{tag}$.
There are two predefined principal ID constants.

Authority ID Constants

$\texttt{top}_{authority}$ – The highest authority in the lattice [Hope never used – would defeat separated privileges. –AMD]
$\texttt{bottom}_{authority}$ – The lowest authority in the lattice

6.1.3 Thread Status

$\mathbf{TS}_{tag}$ – Thread Status tag
TS – Thread Status

The thread status register holds the status flags for the thread. TS==0 is non-error state...

Status Flags

Bit 0–3 – Thread State

- 0x00 – Running
- 0x01 – Runnable
- 0x02 – Uninitialized
- 0x03 – Initialized
- 0x04 – Yield due to timer expiration
- 0x05 – Voluntary yield
- 0x06 – Faulted due to error state
- 0x07 – Halted

Bit 4–7 – Error Code [review: probably one per functional unit –AMD]

- 0x00 – No Error
- 0x01 – TMU Miss
- 0x02 – Stream Blocking
- 0x03 – Arithmetic Error
- 0x04 – Frame Bounds Error
- 0x05 – Atomic Group Error
- 0x06 – Frame Pointer Alignment Error
- 0x07 – Gate Stack Bounds Error
- 0x08 – Transaction Error
- 0x09 – Mangled Thread Frame
- 0x0a – Return Mismatch Error

Bit 8–11 – ALU Status ???

6.1.4 Gate Stack

GS$_{tag}$ – Exit gate stack pointer tag
GS – Exit gate stack pointer

GS is a fat-pointer to the exit gate (continuation) within the gate stack frame for the current procedure being executed.

GS$_{tag}$ may be used for some form of integrity tracking or a similar auditing function. It is included for possible future use, or may be removed if found to be unnecessary.

6.1.5 Interrupt Thread Table

The interrupt thread table **ITT** allows the thread pointers to be registered for each interrupt source. This is per-thread state. Each thread has its own set of trap handlers. This allows threads to be interrupted while in the trap handler state without interfering with other threads. Linearity means we've pulled the thread handler out of the **ITT**, but since this is a per-thread table, that doesn't prevent other threads from finding their trap handlers. It also means many threads can be in their respective trap handlers without interferring with each other—they each have their own context state.

6.1.6 Current Instruction

CI$_{tag}$ – Current instruction tag
CI – Current instruction

CI – represents the currently decoded instruction being executed. The **CI**$_{tag}$ is the tag portion of the instruction word fetched from memory. This register is used in the case of a TMU miss to provide the TMU handler with the instruction causing the TMU miss. Certain cases will cause **CI** to be undefined.

- Prior to first instruction fetch for the thread
- Instruction fetch generates a frame-bounds error

6.1.7 Operand N

The operand registers hold the operands to the currently decoded instruction. These registers are used in the case of a TMU miss to provide the TMU handler with the inputs causing the TMU miss.

OP$^n{}_{tag}$ – Operand n tag
OP^n – Operand n

Different instructions will cause various **OP**^{n} to be undefined, depending upon which operands are used for that instruction.

6.1.8 Memory or Stream Result

MR$_{tag}$ – Memory or Stream Result tag
MR – Memory or Stream Result

MR$_{tag}$ – is an input to the TMU handler routine to check the tag on the atom in memory or coming from a stream read.

MR – Could be left unimplemented

The TMU will have to check **MR**$_{tag}$ for both reads and writes to memory to enforce the "no write down rule". [6/12 – maybe this is going away with the bracket-style register revisions. –AMD]

6.1.9 TMU Result

[With the TMU now producing multiple results, matches less and less with the current design. This is being removed 6/19/12. –AMD]

TR$_{tag}$ – TMU Result tag
TR – TMU Result

~~**TR**$_{tag}$ – is used by the TMU handler routine to return the tag for the result to the thread causing the miss. **TR** – Could be used for TMU status or left unimplemented~~

6.1.10 Non-Cachable TMU Entry

This state is used in cases where we are returning a value through the TMU, but it cannot be used again. When set, it means we should use the TMU entry. **NC** is currently two bits to indicate if the monitor has already fired or if the **NAV** result should be returned.

NC.monitor – monitor has already fired
NC.nav – NAV result is ready to be used

[It's possible there's no need to distinguish between the two cases, so we could just use one bit. –AMD]

In both cases, we put a result in the TMU to allow an operation that caused a trap to complete. If the bit is set, then the TMU result should be used and the bit should be reset. If the bit is not set, then the trap should occur.

- if (TMU.monitor)
  - if (NC.monitor)
    - NC.monitor=false
    - continue operation
  - else
    - take TMU monitor trap
- elseif (TMU.nav_result)
  - if (NC.nav)
    - NC.nav=false
    - continue operation returning NAV from TMU
  - else
    - take NAV trap

6.1.11 faulting Program Counter

fPC$_{tag}$ – faulting Program counter tag
fPC – faulting Program counter

The faulting program counter is a fat pointer that captures the PC of the most recent fault. This may not be the current PC in the case of transactions.

6.1.12 faulting Authority

fA – faulting Authority

fA – represents the authority that the processor was running under when it generated the last fault. This may not be the current A in the case of transactions.

6.1.13 Live Register mask

LR – Live Register Mask

LR – a bit vector representing which of the public general-purpose registers is currently live. The bit vector allows single cycle clearing of a set of registers on a `brtn` or `clearregs`. Any read of a dead (non-live) register returns an Error (NAV). Any write of a register makes it live.

For each register read:

- if (**LR**[*<ri>*]==0)
  - OP*i*=Public Error
- else
  - OP*i*=RF[*<ri>*]

For each register written:

- **LR**[*<ri>*]=1

6.1.14 Writeable Register mask

WR – Writeable Register Mask

WR – a bit vector representing which of the public general-purpose registers can currently be written. The bit vector allows a set of registers to be made read-only or restored to writeable in a single cycle on a `bcall` and `brtn`. Any attempt to write to a read-only (not writeable) register is an error.

How should this error be handled? It could be ignored? (easy) or it could cause a NAV return from some enclosing `bcall` (more complicated).

6.1.15 Instruction Count

Count of logical instructions completed in thread, incremented on instruction commit.

What do about rollover?

6.1.16 Instruction Limit

Limit on logical instructions that can be performed before the next return from a timed call. Each commit checks current count against limit.

6.2 Public State

The public thread state represents the general purpose register resources of the processor. These registers can be freely modified by the code the thread is running.

Public state is **not** modified on an instruction that raises a trap.

Name	Offset from TP_{base}	Description	Update on Faults
GP^0 (EP)	0x00	Environment Pointer	No
GP^1 (FTP)	0x01	Faulting Thread Pointer	No
GP^2	0x02	GP Registers	No
GP^{31}	0x1F	GP Registers	No

6.2.1 Environment Pointer

EP$_{tag}$ – Environment Pointer tag
EP – Environment Pointer

EP provides the environment frame for gate calls. Following a gate call, the **EP** register is updated with the environment frame pointer stored in the gate. Similarly, following a gate return the **EP** is restored to the value from before the gate call.

EP is an alias for register number zero and can be used as a general-purpose register between gate calls.

6.2.2 Faulting Thread Pointer

FTP$_{tag}$ – Faulting Thread Pointer tag
FTP – Faulting Thread Pointer

FTP is used to communicate the thread pointer for the interrupted thread to trap handlers.

FTP is an alias for register number one and can be used as a general-purpose register for threads that are not fault handlers. It can be used as a general-purpose register for fault handlers, but they must be careful to preserve the faulting thread pointer somewhere.

6.2.3 General-Purpose Registers

The general-purpose registers constitute the remainder of the frame. Each frame has an implementation-specific number of registers. We are currently envisioning an implementation with 32 GPRs in a frame.

<r^n>$_{tag}$ – Register n tag
<r^n> – Register n

All instructions refer to registers by number. The register number is the positive offset from the base of the public state within the thread frame. Frame addresses are calculated by adding the size of the private state plus the register number to the **TP**.

6.3 Mangled Thread Frame

As part of our metadata checking, we can validate that the private portion of a thread frame has the correct types when the thread is loaded (runt, resumet, fault). What happens when this check fails?

To first order, this should be an atomic failure of the invoking command. So, if runt calls a thread with a mangled frame, we should fault to the thread calling runt at the runt instruction, calling the mangled thread fault handler. Two potential cases here:

- The thread is mangled, but we can still write into its thread status slot. In this case, we set the thread mangled error in the thread.
- The thread is mangled, preventing the processor from writing into the thread status register. In this case, we place ERROR in **FTP**.

There are two cases for runt: (1) the thread calling runt is prepared for the thread to fail and has code to handle, (2) the thread calling runt is not prepared for the runt to fail. In the first case, the mangled thread handler for the thread calling runt can write state as described above and return to the thread so it can perform its cleanup. In the other case, termination of the runt called thread will also terminate the thread calling runt. The correct thing to do in this case is for the fault handler to update the thread status and perform a `yield2` back to the caller of the thread that called the runt.[2] Current libSafe plan is that only BTS can call runt. BTS must be prepared for the thread it runs to fail with a mangled thread frame.

[2]or is it alwas TTT ? current instruction encoded as TTT.

Similarly, a fault handler invocation that fails because the fault handler thread is mangled should first try to invoke the handler for mangled threads that is in the faulting thread's ITT. If that succeeds it behaves as above, setting thread status and performing a `yield2`. If the attempt to invoke the mangled thread handler fails, then the processor should set **FTP** to ERROR in the timer thread and perform a yield.

A resumet that encouters a mangled thread should similarly first try to set the thread status on the **FTP** and perform a `yield2` back to that thread's caller (probably the BTS via TTT). Failing that it should set **FTP** to ERROR in the thread's caller and perform a `yield2` back to the thread's caller.

[TODO: add more formal statement of behavior. –AMD]

7 Streams

Streams provide a mechanism for serializing access to atoms and can be used for communication between concurrent threads. Streams provide an atomic blocking interface to a shared FIFO data structure. Streams may only have one reader and one writer which reference the stream via stream pointers. The one reader one writer convention must be enforced via the linear pointer discipline.

Streams are buffered so that that reads and writes are decoupled. A stream buffer is allocated as a regular frame of memory. The machine uses atomic groups to ensure that these pointers maintain the following stream invariants. The stream uses Empty_{ag} to denote non-present values in the stream buffer.

7.1 Stream Invariants

- Empty Space in the stream buffer is always Empty_{ag}
- Writes only succeed if the write location is Empty_{ag}
- Reads only succeed if the read location contains a valid atom
- Reads reset the atom to Empty_{ag}
- Read and write pointers are post incremented
- Invalid reads or writes cause an interrupt

ConcreteWare is able to expand buffers if they are full via the stream interrupt handler.

8 TMU

The TMU provides the mechanism to enforce various security properties on executing threads by performing calculations on the tag portion of the atoms. It also provides a mechanism for information flow tracking on values.

[A more up to date description of TMU operation is provided in the libSafe Rule Architecture document. –AMD]

8.1 Rule Checking

The TMU operates as a look up function of many inputs (machine state and inputs for current instruction), returning an allow / disallow flag and the result tags for the different pieces of the machine state.

The TMU may miss when the data for an operation is not in the cache. A TMU miss causes the TMU Miss Handler thread to be activated, which then tries to insert the rule that is needed for the instruction to work.

Allowed operations proceed with the result tag. Disallowed operations cause a gate to the TMU error handler routine.

Inputs to the TMU vary between instructions depending upon number of operands. The set of TMU inputs in the current implementation is as follows (in order):

$\mathbf{TP}_{tag}$ – Current Thread Pointer tag
$\mathbf{CI}_{tag}$ – Current instruction tag
$\mathbf{CI}_{op}$ – Operation Group for current instruction's operation
$\mathbf{PC}_{tag}$ – Program counter tag

A – Authority register
OP$^1{}_{tag}$ – Operand 1 tag (always source 1)
OP$^2{}_{tag}$ – Operand 2 tag (always source 2)
OP$^3{}_{tag}$ – Operand 3 tag (always destination)
MR$_{tag}$ – Memory Result tag

8.2 TMU Rule Interface

The TMU uses a numbered slot interface to load rules into the TMU cache. A complete TMU rule is much wider than the native machine word so it requires a set of load operations to transfer a complete rule into the TMU. The `tmul` instruction is used to transfer one field into a rule slot.

TMU Rule Input Fields

0x00 – **TP**$_{tag}$
0x01 – **CI**$_{tag}$
0x02 – **CI opgroup**
0x03 – **PC**$_{tag}$
0x04 – **A**
0x05 – **OP**$^1{}_{tag}$
0x06 – **OP**$^2{}_{tag}$
0x07 – **OP**$^3{}_{tag}$
0x08 – **MR**$_{tag}$

Any rule field may be a don't-care, which means that particular field does not affect the decision if the operation is allowed or disallowed. Don't cares are currently implemented as a symbol that stays constant throughout. The associated result fields are:

TMU Rule Result Fields

Allow / Disallow (Boolean)
0x09 – **OP**$^3{}_{tag}$
0x0a – **OP**$^2{}_{tag}$
0x0b – **OP**$^1{}_{tag}$
0x0c – **MW**$_{tag}$
0x0d – **TP**$_{tag}$
0x0e – **PC**$_{tag}$
0x0f – **A**

8.3 TMU Miss Handling

A TMU miss occurs when there is no rule loaded in the TMU for the current set of inputs to the TMU. TMU misses are handled by a separate thread. When a miss occurs the TMU thread is swapped in and the **TP** for the thread causing the miss is passed to the TMU handler. The TMU handler routine can inspect the private state of the thread causing the miss to compute the necessary rule for the TMU cache.

8.4 TMU Error Handling

A TMU error occurs when the TMU thread detects a security violation during the processing of a TMU miss. The TMU thread gates to the TMU error handler routine. The error handler can deal with setting status, terminating the thread, and/or returning a NAV.

8.5 TMU Bootstraping

There are a static set of rules for the TMU in order to support concreteware operation. These rules are loaded into a static TMU so they never need to be serviced by a TMU miss. They might be burned into flash on the processor

die or loaded during POR from off chip with a suitable signature check. For development, they are loaded into a designated SRAM before processor boot.

9 Gate Stack

The gate stack holds the return gates associated with gate calls within a thread. A gate call causes the return gate to be pushed on the gate stack. Gate exit causes the return gate to be popped off the gate stack. The gate stack is not directly accessible to running code; it is manipulated implicitly by gate calls and returns and thread switch instructions only.

9.1 Gate Stack Fault

A version is sketched in *fully distributed*.

10 Memory Map

- 0x0000 - 0x003F – Power on thread frame
- POR – Boot loader code (originally around 0x0100, but may be moved back in future so that static authorities can live at fixed addresses that precede the code.)

10.1 Power On Reset

Upon initial start the machine loads the Boot Loader thread with an initial thread frame starting at 0x00 and begins execution at the power on reset location POR as specified as the PC in the Power on thread frame.[3]

The initial thread frame provides a register frame for the bootstrap code.

Initial Machine State

Name	Description	Tag	Atomic Group	Value
TP	Thread Pointer	BootLoader_Private	Thread Pointer	0x0000
QT	Quanta	-	-	-
CLK	Low Counter	-	-	-
SLOT	High Counter	-	-	-
TTT	Timing Trap Thread	-	-	-

Initial Thread Frame

[3]Was originally specified as 0x100, but there is no reason to force it to be there. Discipline is to define a suitable symbol for the POR code and have that loaded into the thread frame. `hardware.asm` defines POR.

Name	Offset from TP_{base}	Description	Tag	Atomic Group	Value
GP^0 (EP)	0x00	Environment Pointer	BootLoader_Private	Frame Pointer	MOAF
GP^1 (FTP)	0x01	Faulting Thread Pointer	-	-	-
GP^2	0x02	GP Registers	Public, Untrusted	Integer	0
GP^3	0x02	GP Registers	-	-	-
GP^{31}	0x1F	GP Registers	-	-	-
ITT^0	0x20	TMU Miss	-	-	-
ITT^1	0x21	Stream Empty/Full?	-	-	-
ITT^2	0x22	Arithmetic Error	-	-	-
ITT^3	0x23	Frame Bounds Error	-	-	-
ITT^4	0x24	Atomic Group Error	-	-	-
ITT^5	0x25	Frame Alignment Error	-	-	-
ITT^6	0x26	Gate Stack Bounds Error	-	-	-
ITT^7	0x27	Transaction Error	-	-	-
ITT^8	0x28	Mangled Thread Error	-	-	-
PC	0x30	Program Counter	BootLoader_Private	Instruction Pointer	POR
A	0x31	Authority	BootLoader	Authority	-
TS	0x32	Thread Status	-	-	-
GS	0x33	Gate Stack pointer	-	-	-
CI	0x34	Current Instruction	-	-	-
OP^1	0x35	OPerand n	-	-	-
OP^2	0x36	OPerand n	-	-	-
OP^3	0x37	OPerand n	-	-	-
MR	0x38	Result of Memory Load	-	-	-
TR	0x39	Result of TMU check	-	-	-
fPC	0x3A	faulting Program Counter	-	-	-
fA	0x3B	faulting Authority	-	-	-

11 Interrupts

Interrupts are implemented with the same **TP** mechanism used for thread switching. When an interrupt occurs the current thread is suspended, without commiting the offending instruction, and the appropriate interrupt thread is loaded into **TP**.

The thread causing the interrupt is placed in the **FTP** register of the handler thread.

11.1 Interrupt Priorities

Interrupts have a static priority ordering fixed in the hardware. Interrupts with a higher priority execute before lower priority ones.

The interrupt priority is listed in the folowing table:

- Timer – highest priority [rationale: timer must take control precisely; we can always come back and handle exceptional conditions on the next scheduling event.]
- Atomic Group Error – priority 2 [if this goes wrong, not clear anything else is meaningful]
- TMU Miss – priority 3 [if operation not allowed, errors irrelevant]
- Frame Pointer Bounds / Alignment Error – priority 4
- Gate Stack Underflow Error – priority 5 (each of these is mutually exclusive)
- Stream Empty/Full? – priority 5
- Arithmetic Error – priority 5
- Transaction Error – priority 5

- Mangled Thread Error – priority 5

TS is encoded to report the highest priority interrupt event that occurs during the cycle.

11.2 Interrupt Semantics

At the end of each operation, the processor effectively performs the following:

Processor State

if (**TS**.ErrorCode==NoError) // no interrupts raised

- // TP remains unchanged
- State changes occur (PC, register file writeback, writes, stream writes and reads, gate stack manipulation)

else

- **fPC**, **fA**, **CI**, **TS**, **OP**^{n}, **MR** are updated based on interrupting operation
  - *N.B.* these values updated based on interrupting operation **are** saved in the case of a transaction abort; this communicates what went wrong in the transaction to the error handler (*e.g.* tells the TMU miss handler what rule is needed).
- No (other) state changes specified by the instruction occur
- if (TID $\neq$ 0) abort_transaction
- **tfTP** $\leftarrow$ **TP** // faulting TP temporary for defining semantics
- if (**TS**.ErrorCode==Timer Interrupt)
  - **TS**.ThreadState=Yield due to timer expiration
  - **TP** $\leftarrow$ **TTT**
  - **TTT** $\leftarrow$ NaV
- else // rest of interrupts
  - **TS**.ThreadState=Faulted due to error state
  - **TP** $\leftarrow$ **ITT**$^{(0xF\&(\mathbf{TS}>>4))}$
  - **ITT**$^{(0xF\&(\mathbf{TS}>>4))}$ $\leftarrow$ NaV // to maintain linearity of TP
- **TP**.**FTP** $\leftarrow$ **tfTP** // write TP of faulting thread into **FTP** of the handler thread

12 Transactions

Maybe rename atomic block

Transactions allow a set of operations to occur atomically. Either all operations in the transaction complete without interleaving or none of them complete. An aborted transaction restarts at the beginning of the transaction when resumed. Initially, this is used to maintain atomicity in the face of interrupts and traps (timer, TMU, GC, etc.). Long term, this could also support atomicity with respect to concurrent operation.

Transactions will be limited in size. They will support a maximum number of total instructions (TXN_MAXINSTR), maximum number of reads (TXN_MAXREAD), and a maximum number of writes (TXN_MAXWRITE) TXN_MAXWRITE is needed to bound the size of the write buffer. For true concurrent, shared-memory operations, we must keep track of the reader set to detect any writes that might invalidate the transaction; TXN_MAXREAD specifies how large of a reader set we will support. We expect these to be small (*e.g.* TXN_MAXWRITE 4–16, TXN_MAXREAD 32–1024), but need more experience with Concreteware before selecting specific values. TXN_MAXINSTR sets the maximum number of complete rules (all fields) that the TMU transaction buffer can handle.

When a transaction starts with a transbegin, the thread state is all written out to memory. Execution continues with no state going to memory or the TMU. Stream operations are allowed during a transaction.[4] Writes and TMU changes during a transaction are buffered. Thread Frame state is modified only on the processor code. If

[4]Stream operations were originally disallowed. We now [3/12] propose to allow.

the transaction completes, it writes all state to memory and TMU without interruption (Thread Frame, flush write buffer). If the transaction is aborted before completion, the memory state properly represents the state of the thread, except for the private state that represents the trap condition; the trap-condition private state is written to memory and the rest of the in-processor state is discarded.

Current plan is that gate calls (any calls) are not allowed in a transaction. Looping is not allowed in transaction. This will likely show up as not allowing backward control flow transfers (lower PC in frame) while executing the transaction.

12.1 Transaction Abort

abort_transaction:

TID ← 0;
// **fPC**, **fA**, **CI**, **TS**, **OP**^{n}, **MR** are updated in any interrupt case, including this one
flush write buffer and transaction read tags

13 Forwarding Pointers

If the atomic group of a dereferenced register pointer used for a read is a Forwarding Pointer, the register payload should be replaced with the forwarding pointer payload and the instruction reissued. If the **PC** target is a Forwarding Pointer, the **PC** value should be updated and the instruction reissued at fetch.

13.1 PC Behavior

- if (mem[**PC**]$_{ag}$==Forwarding Pointer)
  - **PC**=mem[**PC**] // *n.b.* ***PC***$_{ag}$ *remains unchanged*
  - treat current instruction (that did not read) as a NOOP
  - // do not incremental **PC**; so re-issue fetch with the updated **PC** on the next cycle
- else
  - continue with normal instruction semantics

13.2 Memory Dereference Behavior

- if (**MR**$_{ag}$==Forwarding Pointer)
  - turn instruction into a register assignment: pointer_register(instr)[5]← **MR** // *n.b. atomic group remains unchanged in register*
  - // ok to write to a read-only register in this case, since semantically not changing
  - // do not incremental **PC**; so re-issue fetch with the updated **PC** on the next cycle
- else
  - continue with normal instruction semantics

Effected instructions: mvmr, cpmr, *call (incl. *jmp), grtn (?), str?, cp{ar,ht,io,mt}, mvrm, cprm, stw?

REVIEW: does this need to check **GS**, or invariants to guarantee that's ok?

14 Arithmetic OP Codes

Here is a minimum selection of arithmetic op-codes. The full set will be added later.

[5]unfortunately, pointers don't currently always come from the same register. It is *<r1>* for read and gate operations; **GS** for grtn; *<r2>* for writes (mvrm, cprm and stream writes)

14.1 Operand Conventions

This document makes a distinction between register numbers, offsets and addresses.

Offsets relative to the current $\mathbf{PC}_{base}$ are unsigned and used for branching or load constant instructions.

Register numbers are unsigned integer offsets within the public portion of the thread frame used for accessing instruction operands.

Addresses are simply fat pointers which encode both the frame and location within the frame.

14.1.1 `add <r1> <r2> <r3>` – Add

Add the Integer_{ag} atom in *<r1>* to *<r2>*, result in *<r3>*

Operands

<r1> – Source 1
<r2> – Source 2
<r3> – Destination

Allowed Operand Groups

$\textit{<r1>}_{ag} = \text{Integer}_{ag}$, $\textit{<r2>}_{ag} = \text{Integer}_{ag}$

Processor State

$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$
$\textit{<r3>}_{tag} \leftarrow \mathbf{TMU}$
<r3> $\leftarrow$ *<r1>* + *<r2>*
$\textit{<r3>}_{ag} \leftarrow \textit{<r1>}_{ag}$

At some point, we should be clear about arithmetic instructions that overflow versus the ones that perform modulo arithmetic. Here's some instructions to consider:

- addm s r1 r2 r3: r3=(r1+r2)&(2^s-1) [same signed/unsigned]
- addb s r1 r2 r3: t=r2+r3; if overflow s bits, branch to code pointer r1 else r3=t
- addc s r1 r2 r3: t=r1+r2 r3=t&(2^s-1) r2=(r1+r2)>>s [exploit 2 writeback ports]

In all cases s is an immediate field specifying the width of the instruction. Perhaps this is too general. I believe it subsumes signed/unsigned.

14.1.2 `addf <r1> <r2> <r3>` – Double-Precision Floating-Point Add

Add the Double_{ag} atom in *<r1>* to *<r2>*, result in *<r3>*

Operands

<r1> – Source 1
<r2> – Source 2
<r3> – Destination

Allowed Operand Groups

$\textit{<r1>}_{ag} = \text{Double}_{ag}$, $\textit{<r2>}_{ag} = \text{Double}_{ag}$

Processor State

PC $\leftarrow$ **PC** + **1**
<r3>$_{tag}$ $\leftarrow$ **TMU**
<r3> $\leftarrow$ <r1> $+_{double}$ <r2>
<r3>$_{ag}$ $\leftarrow$ <r1>$_{ag}$

Will need fixed to/from floating-point instructions

14.1.3 `sub <r1> <r2> <r3>` – Subtract

Subtract the atom in <r2> from <r1>, result in <r3>

Operands

<r1> – Source 1
<r2> – Source 2
<r3> – Destination

Allowed Operand Groups

<r1>$_{ag}$ = Integer$_{ag}$, <r2>$_{ag}$ = Integer$_{ag}$

Processor State

PC $\leftarrow$ **PC** + **1**
<r3>$_{tag}$ $\leftarrow$ **TMU**
<r3> $\leftarrow$ <r1> - <r2>
<r3>$_{ag}\leftarrow$ <r1>$_{ag}$

14.1.4 `mul <r1> <r2> <r3>` – Multiply

Multiply the atom in <r2> with <r1>, result in <r3>

Operands

<r1> – Source 1
<r2> – Source 2
<r3> – Destination

Allowed Operand Groups

<r1>$_{ag}$ = Integer$_{ag}$, <r2>$_{ag}$ = Integer$_{ag}$

Processor State

PC $\leftarrow$ **PC** + **1**
<r3>$_{tag}$ $\leftarrow$ **TMU**
<r3> $\leftarrow$ <r1> * <r2>
<r3>$_{ag}\leftarrow$ <r1>$_{ag}$

TMU rules are expected to enforce <r1>'s label is joined with <r2>'s label and the **PC**'s label to create the label for <r3>.

14.1.5 `and <r1> <r2> <r3>` – Bitwise And

Bitwise and the atom in *<r1>* with *<r2>*, result in *<r3>*

Operands

<r1> – Source 1
<r2> – Source 2
<r3> – Destination

Allowed Operand Groups

$\textit{<r1>}_{ag} = \text{Integer}_{ag}$, $\textit{<r2>}_{ag} = \text{Integer}_{ag}$

Processor State

PC $\leftarrow$ **PC** + **1**
$\textit{<r3>}_{tag} \leftarrow$ **TMU**
<r3> $\leftarrow$ *<r1>* & *<r2>*
$\textit{<r3>}_{ag} \leftarrow \text{Integer}_{ag}$

TMU rules are expected to enforce *<r1>*'s label is joined with *<r2>*'s label and the **PC**'s label to create the label for *<r3>*.

14.1.6 `or <r1> <r2> <r3>` – Bitwise Or

Bitwise or the atom in *<r1>* with *<r2>*, result in *<r3>*

Operands

<r1> – Source 1
<r2> – Source 2
<r3> – Destination

Allowed Operand Groups

$\textit{<r1>}_{ag} = \text{Integer}_{ag}$, $\textit{<r2>}_{ag} = \text{Integer}_{ag}$

Processor State

PC $\leftarrow$ **PC** + **1**
$\textit{<r3>}_{tag} \leftarrow$ **TMU**
<r3> $\leftarrow$ *<r1>* | *<r2>*
$\textit{<r3>}_{ag} \leftarrow \text{Integer}_{ag}$

TMU rules are expected to enforce *<r1>*'s label is joined with *<r2>*'s label and the **PC**'s label to create the label for *<r3>*.

14.1.7 `xor <r1> <r2> <r3>` – Bitwise Xor

Bitwise xor the atom in *<r1>* with *<r2>*, result in *<r3>*

Operands

<r1> – Source 1
<r2> – Source 2
<r3> – Destination

Allowed Operand Groups

$\textit{<r1>}_{ag} = \text{Integer}_{ag}$, $\textit{<r2>}_{ag} = \text{Integer}_{ag}$

Processor State

$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$
$\textit{<r3>}_{tag} \leftarrow \mathbf{TMU}$
$\textit{<r3>} \leftarrow \textit{<r1>} \oplus \textit{<r2>}$
$\textit{<r3>}_{ag} \leftarrow \text{Integer}_{ag}$

TMU rules are expected to enforce *<r1>*'s label is joined with *<r2>*'s label and the **PC**'s label to create the label for *<r3>*.

14.1.8 `not <r1> <r3>` – Bitwise Inversion (Logical Complement)

Bitwise invert the atom in *<r1>* and put result in *<r3>*

Operands

<r1> – Source 1
<r3> – Destination

Allowed Operand Groups

$\textit{<r1>}_{ag} = \text{Integer}_{ag}$

Processor State

$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$
$\textit{<r3>}_{tag} \leftarrow \mathbf{TMU}$
$\textit{<r3>} \leftarrow \overline{\textit{<r1>}}$
$\textit{<r3>}_{ag} \leftarrow \text{Integer}_{ag}$

14.1.9 `shl <r1> <r2> <r3>` – Shift Left

Bitwise left shift the atom in *<r1>* by *<r2>* bits, result in *<r3>*

Operands

<r1> – Source 1
<r2> – Source 2
<r3> – Destination

Allowed Operand Groups

$\texttt{<r1>}_{ag}$ = Integer$_{ag}$, $\texttt{<r2>}_{ag}$ = Integer$_{ag}$

Processor State

PC $\leftarrow$ **PC + 1**
$\texttt{<r3>}_{tag} \leftarrow$ **TMU**
<r3> $\leftarrow$ <r1> << <r2>
$\texttt{<r3>}_{ag} \leftarrow$ Integer$_{ag}$

TMU rules are expected to enforce <r1>'s label is joined with <r2>'s label and the **PC**'s label to create the label for <r3>.

14.1.10 `shr <r1> <r2> <r3>` – Shift Right

Bitwise right shift the atom in <r1> by <r2> bits, result in <r3>

Operands

<r1> – Source 1
<r2> – Source 2
<r3> – Destination

Allowed Operand Groups

$\texttt{<r1>}_{ag}$ = Integer$_{ag}$, $\texttt{<r2>}_{ag}$ = Integer$_{ag}$

Processor State

PC $\leftarrow$ **PC + 1**
$\texttt{<r3>}_{tag} \leftarrow$ **TMU**
<r3> $\leftarrow$ <r1> >> <r2>
$\texttt{<r3>}_{ag} \leftarrow$ Integer$_{ag}$

TMU rules are expected to enforce <r1>'s label is joined with <r2>'s label and the **PC**'s label to create the label for <r3>.

14.1.11 `test <r1> <r2> <r3>` – Test Equality

Test the atom in <r1> with atom in <r2>, result in <r3>

Operands

<r1> – Source 1
<r2> – Source 2
<r3> – Destination

Allowed Operand Groups

$\texttt{<r1>}_{ag} = \texttt{<r2>}_{ag}$

Processor State

PC ← **PC** + **1**
<r3>$_{tag}$ ← **TMU**
<r3>← **1** : *<r1>* == *<r2>*
<r3>← **0** : *<r1>* ≠ *<r2>*
<r3>$_{ag}$← Integer$_{ag}$

TMU rules are expected to enforce *<r1>*'s label is joined with *<r2>*'s label and the **PC**'s label to create the label for *<r3>*.

14.1.12 `testgrp <r1> <r2> [grpid]` – Test Particular Group

Test the atom in *<r1>* with group *[grpid]*, result in *<r2>*

Operands

<r1> – Source 1
<r2> – Destination
[grpid] –group id to match

Allowed Operand Groups

<r1> – any

Processor State

PC ← **PC** + **1**
<r2>$_{tag}$ ← **TMU**
<r2>$_{ag}$← Integer$_{ag}$
<r2> ← **1** : *<r1>* == *[grpid]*
<r2> ← **0** : *<r1>* ≠ *[grpid]*

15 Control Flow OP Codes

Control flow op-codes are divided into frame local and inter-frame versions. Frame local control flow uses offsets from the base of the PC to specify control flows within a frame. Intra-frame control flow uses addresses specified as fat pointers for the destinations.

Frame boundaries for **PC** relative offsets are bounds checked against the frame encoded by the **PC** fat pointer.

Inter-frame instructions specify a fat pointer address for control transfers. This allows control flow change to any frame the code holds a pointer to.

15.1 Frame Local Control Flow

Frame local instructions represent simple control flow transfers without any change to the **PC**$_{tag}$. Conditional branch instructions join the taint from the condition to the control flow taint. There is no scoping of the taint as it accumulates.

15.1.1 `jmp [offset]` – Unconditional Jump

Unconditional jump to **PC**$_{base}$ + *[offset]*

Operands

[offset] – Unsigned offset from current code frame base

Processor State

PC $\leftarrow$ **PC**$_{base}$ + *[offset]*

TMU rules are expected to have no effect on the **PC**'s label.

15.1.2 `beq <r1> [offset]` – Branch Equal

Branch to **PC**$_{base}$ + *[offset]* if *<r1>* = **0**

Operands

<r1> – Atom to test
[offset] – Branch destination as unsigned offset from current code frame base

Allowed Operand Groups

<r1>$_{ag}$ = Integer$_{ag}$

Processor State

PC$_{tag}$ $\leftarrow$ **TMU**
PC $\leftarrow$ **PC** + **1**: *<r1>* $\neq$ **0**
PC $\leftarrow$ **PC**$_{base}$ + *[offset]* : *<r1>* = **0**

TMU rules are expected to enforce *<r1>*'s label is joined with the **PC**'s label to create the new label for **PC**.

15.1.3 `bne <r1> [offset]` – Branch Not Equal

Branch to **PC**$_{base}$ + *[offset]* if *<r1>* $\neq$ **0**

Operands

<r1> – Atom to test
[offset] – Branch destination as unsigned offset from current code frame base

Allowed Operand Groups

<r1>$_{ag}$ = Integer$_{ag}$

Processor State

PC$_{tag}$ $\leftarrow$ **TMU**
PC $\leftarrow$ **PC** + **1**: *<r1>* = **0**
PC $\leftarrow$ **PC**$_{base}$ + *[offset]* : *<r1>* $\neq$ **0**

TMU rules are expected to enforce *<r1>*'s label is joined with the **PC**'s label to create the new label for **PC**.

15.1.4 `bneg <r1> [offset]` – Branch Negative

Branch to $\mathbf{PC}_{base}$ + *[offset]* if *<r1>* $< \mathbf{0}$

Operands

<r1> – Atom to test
[offset] – Branch destination as unsigned offset from current code frame base

Allowed Operand Groups

$\textit{<r1>}_{ag} = \text{Integer}_{ag}$

Processor State

$\mathbf{PC}_{tag} \leftarrow \mathbf{TMU}$
$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$: *<r1>* $\geq \mathbf{0}$
$\mathbf{PC} \leftarrow \mathbf{PC}_{base} +$ *[offset]* : *<r1>* $< \mathbf{0}$

TMU rules are expected to enforce *<r1>*'s label is joined with the **PC**'s label to create the new label for **PC**.

15.1.5 `bpos <r1> [offset]` – Branch Positive

Branch to $\mathbf{PC}_{base}$ + *[offset]* if *<r1>* $> \mathbf{0}$

Operands

<r1> – Atom to test
[offset] – Branch destination as unsigned offset from current code frame base

Allowed Operand Groups

$\textit{<r1>}_{ag} = \text{Integer}_{ag}$

Processor State

$\mathbf{PC}_{tag} \leftarrow \mathbf{TMU}$
$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$: *<r1>* $\leq \mathbf{0}$
$\mathbf{PC} \leftarrow \mathbf{PC}_{base} +$ *[offset]* : *<r1>* $> \mathbf{0}$

TMU rules are expected to enforce *<r1>*'s label is joined with the **PC**'s label to create the new label for **PC**.

15.2 Inter-frame Control Flow

Inter-frame instructions provide control flow between frames. These instructions use fat pointers to refer to the call site.

15.2.1 `fjmp <r1>` – Frame Jump

Unconditional jump to frame address

Operands

<r1> – Instruction pointer to jump to

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = InstructionPointer$_{ag}$

Processor State

PC$_{tag}$ ← **TMU**
PC ← *<r1>*

Error Conditions

$\textit{<r1>}_{ag} \neq$ Instruction Pointer

15.2.2 `gcall <r1>` – Procedure call with authority change

Invoke gate at address in *<r1>* and push return gate on gate stack.

Operands

<r1> – Address of gate

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = Gate Pointer$_{ag}$

Processor State

GS ← **GS** + 4
GS.PC$_{tag}$ ← **PC**$_{tag}$
GS.PC ← **PC+1**
GS.A$_{tag}$ ← **A**$_{tag}$
GS.A ← **A**
GS.EP$_{tag}$ ← **EP**$_{tag}$
GS.EP ← **EP**
GS.R$_{ag}$ ← **Error**$_{ag}$ // unused, distinguish from bcall record
PC$_{tag}$ ← **TMU**
PC ← *<r1>*.**PC**
A$_{tag}$ ← *<r1>*.**A**$_{tag}$
A ← *<r1>*.**A**
EP$_{tag}$ ← *<r1>*.**EP**$_{tag}$
EP ← *<r1>*.**EP**
MR$_{tag}$ ← *<r1>*.**PC**$_{tag}$
MR$_{ag}$ ← *<r1>*.**PC**$_{ag}$
MR ← *<r1>*.**PC**

Error Conditions

<r1>$_{ag} \neq$ Gate Pointer
<r1>.**A** $_{ag} \neq$ Authority
<r1>.**PC**$_{ag} \neq$ Instruction Pointer

TMU rules are expected to enforce caller's **PC** label is joined with the **PC** label in the gate.

15.2.3 `gfcall <r1> <r2> <r3> [mask]` – Cacheable procedure call with authority change

Invoke gate at address in <r1> on <r2> and <r3> and put result in <r3>.
[bgfcall may make more sense than this...so maybe only do that? –AMD]

Operands

<r1> – Address of gate
<r2> – Argument
<r3> – Argument and Destination Register
[mask] – top two bits specify how to mask the input (00 – use all; 01 – mask out value for <r3>; 10 – mask out <r3> entirely; 11 – mask out <r3> and value on <r2>)

Allowed Operand Groups

<r1>$_{ag}$ = Gate Pointer$_{ag}$

Processor State: gfcache miss

GS ← **GS** + **4**
GS.PC$_{tag}$ ← **PC**$_{tag}$
GS.PC ← **PC+1**
GS.A$_{tag}$ ← **A**$_{tag}$
GS.A ← **A**
GS.EP$_{tag}$ ← **EP**$_{tag}$
GS.EP ← **EP**
GS.R$_{ag}$ ← **Error**$_{ag}$ // unused, distinguish from bcall record
PC$_{tag}$ ← **TMU**
PC ← <r1>.**PC**
A$_{tag}$ ← <r1>.**A**$_{tag}$
A ← <r1>.**A**
EP$_{tag}$ ← <r1>.**EP**$_{tag}$
EP ← <r1>.**EP**
MR$_{tag}$ ← <r1>.**PC**$_{tag}$
MR$_{ag}$ ← <r1>.**PC**$_{ag}$
MR ← <r1>.**PC**

Processor State: gfcache hit

PC$_{tag}$ ← **TMU**[Revist as work out bcall. Issue: in non-restoring gcall world, the **PC**$_{tag}$ might change. Maybe the gfcall should be a bgfcall... –AMD]
PC ← **PC** + **1**
<r3> ← cached result of gcall <r1> on <r2> and <r3>
<r3>$_{ag}$ ← atomic group from cached result of gcall <r1> on <r2> and <r3>

<r3>$_{tag}$ ← **TMU**

Error Conditions

<r1>$_{ag} \neq$ Gate Pointer
<r1>.**A** $_{ag} \neq$ Authority
<r1>.**PC**$_{ag} \neq$ Instruction Pointer
mask specified in *[mask]* [9:8] does not match mask read from gfcache

TMU rules are expected to enforce caller's **PC** label is joined with the **PC** label in the gate.

15.2.4 `gjmp <r1>` – Gate jump without gate return

Invoke gate at address in *<r1>* without push return gate on gate stack.

Operands

<r1> – Address of gate

Allowed Operand Groups

<r1>$_{ag}$ = GatePointer$_{ag}$

Processor State

PC$_{tag}$ ← **TMU**
PC ← *<r1>*.**PC**
A$_{tag}$ ← *<r1>*.**A**$_{tag}$
A ← *<r1>*.**A**
EP$_{tag}$ ← *<r1>*.**EP**$_{tag}$
EP ← *<r1>*.**EP**
MR$_{tag}$ ← *<r1>*.**PC**$_{tag}$
MR$_{ag}$ ← *<r1>*.**PC**$_{ag}$
MR ← *<r1>*.**PC**

Error Conditions

<r1>$_{ag} \neq$ Gate Pointer
<r1>.**A**$_{ag} \neq$ Authority
<r1>.**PC**$_{ag} \neq$ Instruction Pointer

TMU rules are expected to enforce caller's **PC** label is joined with the **PC** label in the gate.

15.2.5 `grtn` – Return from Gate Call

Return from a gate call

Operands

None

Allowed Operand Groups

None

Processor State

PC$_{tag}$ ← **TMU**
MR$_{ag}$ ← **GS.PC**$_{ag}$
MR$_{tag}$ ← **GS.PC**$_{tag}$
MR ← **GS.PC**
PC ← **GS.PC**
A ← **GS.A**
EP$_{tag}$ ← **GS.EP**$_{tag}$
EP ← **GS.EP**
GS ← **GS - 4**

Error Conditions

GS.**A**$_{ag}$ ≠ Authority
GS.**PC**$_{ag}$ ≠ Instruction Pointer
GS.**R**$_{ag}$ = Integer$_{ag}$ // will be Integer$_{ag}$ for bcall record

15.2.6 `call <r1>` – Procedure Call, no authority change

Call procedure in *<r1>*. Builds a return gate with the current authority and environment and pushes on gate stack.

Operands

<r1> – Pointer to procedure

Allowed Operand Groups

<r1>$_{ag}$ = Instruction Pointer$_{ag}$

Processor State

GS ← **GS** + **4**
GS.PC$_{tag}$ ← **PC**$_{tag}$
GS.PC ← **PC+1**
GS.A$_{tag}$ ← **A**$_{tag}$
GS.A ← **A**
GS.EP$_{tag}$ ← **EP**$_{tag}$
GS.EP ← **EP**
GS.R$_{ag}$ ← **Error**$_{ag}$ // unused, distinguish from bcall record
PC$_{tag}$ ← **TMU**
EP← Uninitialized // *not coming form gate closure – **EP** is for the* lexically *enclosing environment, not the callers environment. So, not appropriate to chain through **EP**.*
PC ← *<r1>*

Error Conditions

<r1>$_{ag}$ ≠ Instruction Pointer

15.2.7 `gacall <r1> <r2>` – Procedure call with authority change and augmentation

Invoke gate at address in *<r1>* with its authority extended by first-class authority in *<r2>* and push return gate on gate stack.

Operands

<r1> – Address of gate
<r2> – Additional Authority to use with gate

Allowed Operand Groups

$\langle r1 \rangle_{ag}$ = Gate Pointer$_{ag}$, $\langle r2 \rangle_{ag}$ = Authority$_{ag}$

Processor State

GS ← **GS** + 4
GS.PC$_{tag}$ ← **PC**$_{tag}$
GS.PC ← **PC+1**
GS.A$_{tag}$ ← **A**$_{tag}$
GS.A ← **A**
GS.EP$_{tag}$ ← **EP**$_{tag}$
GS.EP ← **EP**
GS.R$_{ag}$ ← **Error**$_{ag}$ // unused, distinguish from bcall record
PC$_{tag}$ ← **TMU**
PC ← *<r1>*.**PC**
A ← *<r1>*.**A** with *<r2>* // from TMU output
EP$_{tag}$ ← *<r1>*.**EP**$_{tag}$
EP ← *<r1>*.**EP**
MR$_{ag}$ ← *<r1>*.**A**$_{ag}$
MR$_{tag}$ ← *<r1>*.**A**
// Perhaps a subtlety – in this case the TMU's M.MR gets the A payload not the A.tag; whereas normally TMU.MR gets MR.tag
// what actually happens is the full record (PC,EP,A) is coming in from memory, but is wider than a single atom (3 atoms), but only the A piece is going to the TMU through the TMU MR port
// Also note that <r2> payload needs to be seen by the TMU. 4/20/12 working model is to feed that into the $\langle r3 \rangle_{tag}$ *input to the TMU.*

Error Conditions

$\langle r1 \rangle_{ag} \neq$ Gate Pointer
$\langle r2 \rangle_{ag} \neq$ Authority
<r1>.**A**$_{ag} \neq$ Authority
<r1>.**PC**$_{ag} \neq$ Instruction Pointer

TMU rules are expected to enforce caller's **PC** label is joined with the **PC** label in the gate.

15.2.8 `gajmp <r1>` – Jump with authority change and augmentation without gate return

Invoke gate at address in *<r1>* with its authority extended by the authority in *<r2>* without push return gate on gate stack.

Operands

<r1> – Address of gate
<r2> – Additional authority to use with jump

Allowed Operand Groups

$\texttt{<r1>}_{ag}$ = GatePointer$_{ag}$, $\texttt{<r2>}_{ag}$ = Authority$_{ag}$

Processor State

PC$_{tag}$ ← **TMU**
PC ← <r1>**.PC**
A ← <r1>**.A** with <r2> // from TMU output
EP$_{tag}$ ← <r1>**.EP**$_{tag}$
EP ← <r1>**.EP**
MR$_{ag}$ ← <r1>.**A**$_{ag}$
MR$_{tag}$ ← <r1>.**A**
// Perhaps a subtlety – in this case the TMU's M.MR gets the A payload not the A.tag; whereas normally TMU.MR gets MR.tag
// what actually happens is the full record (PC,EP,A) is coming in from memory, but is wider than a single atom (3 atoms), but only the A piece is going to the TMU through the TMU MR port
// Also note that <r2> payload needs to be seen by the TMU. 4/20/12 working model is to feed that into the $\texttt{<r3>}_{tag}$ *input to the TMU.*

Error Conditions

$\texttt{<r1>}_{ag} \neq$ Gate Pointer
$\texttt{<r2>}_{ag} \neq$ Authority
<r1>.**A**$_{ag} \neq$ Authority
<r1>.**PC**$_{ag} \neq$ Instruction Pointer

TMU rules are expected to enforce caller's **PC** label is joined with the **PC** label in the gate.

15.2.9 `acall <r1> <r2>` – Procedure Call with caller specified authority

Call procedure in <r1> and change authority to <r2>. Builds a return record with the current authority and environment and pushes on gate stack.

Operands

<r1> – Pointer to procedure
<r2> – Authority to use with gate

Allowed Operand Groups

$\texttt{<r1>}_{ag}$ = Instruction Pointer$_{ag}$, $\texttt{<r2>}_{ag}$ = Authority$_{ag}$

Processor State

GS ← **GS** + **4**
GS.PC$_{tag}$ ← **PC**$_{tag}$
GS.PC ← **PC+1**
GS.A$_{tag}$ ← **A**$_{tag}$

GS.A $\leftarrow$ **A**
GS.EP$_{tag}$ $\leftarrow$ **EP**$_{tag}$
GS.EP $\leftarrow$ **EP**
GS.R$_{ag}$ $\leftarrow$ **Error**$_{ag}$ // unused, distinguish from bcall record
PC$_{tag}$ $\leftarrow$ **TMU**
A $\leftarrow$ *<r2>*
EP$\leftarrow$ Uninitialized // *not coming form gate closure; see full comment on* `call`
PC $\leftarrow$ *<r1>*

Error Conditions

<r1>$_{ag} \neq$ Instruction Pointer
<r2>$_{ag} \neq$ Authority

15.2.10 `lcall <r1> <r2>` – Procedure Call with caller specified reduction in authority

Call procedure in *<r1>* and lower current authority by *<r2>*. Builds a return record with the current authority and environment and pushes on gate stack. Only difference from `acall` is rules applied by TMU.

Operands

<r1> – Pointer to procedure
<r2> – Authority to use with gate

Allowed Operand Groups

<r1>$_{ag}$ = Instruction Pointer$_{ag}$, *<r2>*$_{ag}$ = Authority$_{ag}$

Processor State

GS $\leftarrow$ **GS** + **4**
GS.PC$_{tag}$ $\leftarrow$ **PC**$_{tag}$
GS.PC $\leftarrow$ **PC+1**
GS.A$_{tag}$ $\leftarrow$ **A**$_{tag}$
GS.A $\leftarrow$ **A**
GS.EP$_{tag}$ $\leftarrow$ **EP**$_{tag}$
GS.EP $\leftarrow$ **EP**
GS.R$_{ag}$ $\leftarrow$ **Error**$_{ag}$ // unused, distinguish from bcall record
PC$_{tag}$ $\leftarrow$ **TMU**
A $\leftarrow$ *<r2>*
EP$\leftarrow$ Uninitialized // *not coming form gate closure; see full comment on* `call`
PC $\leftarrow$ *<r1>*

Error Conditions

<r1>$_{ag} \neq$ Instruction Pointer
<r2>$_{ag} \neq$ Authority

15.2.11 `rcall <r1> <r2>` – Procedure Call with caller specified additional authority

Call procedure in *<r1>* and raise current authority by *<r2>*. Builds a return record with the current authority and environment and pushes on gate stack. Only difference from `acall` is rules applied by TMU.

Operands

<r1> – Pointer to procedure
<r2> – Authority to use with gate

Allowed Operand Groups

$\langle r1 \rangle_{ag}$ = Instruction Pointer$_{ag}$, $\langle r2 \rangle_{ag}$ = Authority$_{ag}$

Processor State

GS ← **GS** + 4
GS.PC$_{tag}$ ← **PC**$_{tag}$
GS.PC ← **PC+1**
GS.A$_{tag}$ ← **A**$_{tag}$
GS.A ← **A**
GS.EP$_{tag}$ ← **EP**$_{tag}$
GS.EP ← **EP**
GS.R$_{ag}$ ← **Error**$_{ag}$ // unused, distinguish from bcall record
PC$_{tag}$ ← **TMU**
A ← *<r2>*
EP← Uninitialized // *not coming form gate closure; see full comment on* `call`
PC ← *<r1>*

Error Conditions

$\langle r1 \rangle_{ag} \neq$ Instruction Pointer
$\langle r2 \rangle_{ag} \neq$ Authority

15.2.12 `bcall <r1> <r3> [result-reg]` – Bracket Call

Invokes the bracket at *<r1>* with the registers specified by *<r3>* as writeable, expecting the result to go in register *[result-reg]* which will have a tag $\langle r3 \rangle_{tag}$.

Operands

<r1> – Address of procedure to call
<r2> – *unused*
<r3> – bit vector specifying which registers will be writeable during the call; except for the designated return register, these will be cleared upon `brtn`; tagged with tag equal to tag for result
[result-reg] – specify register that should get result and be tagged by $\langle r3 \rangle_{tag}$

Allowed Operand Groups

$\langle r1 \rangle_{ag}$ = Instruction Pointer$_{ag}$, $\langle r3 \rangle_{ag}$ = Integer$_{ag}$

Processor State

GS ← **GS** + **4**
GS.PC$_{tag}$ ← **PC**$_{tag}$
GS.PC ← **PC+1**
GS.A$_{tag}$ ← **A**$_{tag}$
GS.A ← **A**
GS.EP$_{tag}$ ← **EP**$_{tag}$
GS.EP ← **EP**
GS.R$_{ag}$ ← **Integer**$_{ag}$
GS.R$_{tag}$ ← *<r3>*$_{tag}$
GS.R ← **(WR<<32)**|*[result-reg]* // *mixing tag, mask, and register in one atom to make compact*
PC$_{tag}$ ← **TMU**
PC ← *<r1>*
EP← Uninitialized // *not coming form gate closure; see call*
WR=**WR** & *<r3>*

Error Conditions

<r1>$_{ag}$ ≠ Instruction Pointer$_{ag}$
<r3>$_{ag}$ ≠ Integer$_{ag}$
<r3>[*[result-reg]*] ≠ 1 // result register should be writeable

15.2.13 `bgcall <r1> <r3> [result-reg]` – Bracket call with authority change

Invokes the bracket gate at *<r1>* with the registers specified by *<r3>* as writeable, expecting the result to go in register *[result-reg]* which will have a tag *<r3>*$_{tag}$.

Operands

<r1> – Gate to call
<r2> – *unused*
<r3> – bit vector specifying which registers will be writeable during the call; except for the designated return register, these will be cleared upon `brtn`; tagged with tag equal to tag for result
[result-reg] – specify register that should get result and be tagged by *<r3>*$_{tag}$

Allowed Operand Groups

<r1>$_{ag}$ = Gate Pointer$_{ag}$, *<r3>*$_{ag}$ = Integer$_{ag}$

Processor State

GS ← **GS** + **4**
GS.PC$_{tag}$ ← **PC**$_{tag}$
GS.PC ← **PC+1**
GS.A$_{tag}$ ← **A**$_{tag}$
GS.A ← **A**
GS.EP$_{tag}$ ← **EP**$_{tag}$
GS.EP ← **EP**
GS.R$_{ag}$ ← **Integer**$_{ag}$
GS.R$_{tag}$ ← *<r3>*$_{tag}$
GS.R ← **(WR<<32)**|*[result-reg]* // *mixing tag, mask, and register in one atom to make compact*
PC$_{tag}$ ← **TMU**

PC ← *<r1>*.**PC**
$\mathbf{A}_{tag}$ ← *<r1>*.$\mathbf{A}_{tag}$
A ← *<r1>*.**A**
$\mathbf{EP}_{tag}$ ← *<r1>*.$\mathbf{EP}_{tag}$
EP ← *<r1>*.**EP**
$\mathbf{MR}_{tag}$ ← *<r1>*.$\mathbf{PC}_{tag}$
$\mathbf{MR}_{ag}$ ← *<r1>*.$\mathbf{PC}_{ag}$
MR ← *<r1>*.**PC**
WR=**WR** & *<r3>*

Error Conditions

<r1>$_{ag}$ ≠ Gate Pointer$_{ag}$
<r3>$_{ag}$ ≠ Integer$_{ag}$
<r3>[*[result-reg]*] ≠ 1 // result register should be writeable
<r1>.**A** $_{ag}$≠ Authority$_{ag}$
<r1>.$\mathbf{PC}_{ag}$ ≠ Instruction Pointer$_{ag}$

15.2.14 `bgfcall <r1> <r2> <r3> [mask]` – Cacheable bracket procedure call with authority change

Invoke gate at address in *<r1>* on *<r2>* and *<r3>* and put result in *<r3>*.

Operands

<r1> – Gate Pointer
<r2> – Argument
<r3> – Argument and Destination Register tagged as destination register should be tagged
[mask] – Mask of arguments and writable registers; top two bits specify argument mask (00 – use all; 01 – mask out value for *<r3>*; 10 – mask out *<r3>* entirely; 11 – mask out *<r3>* and value on *<r2>*); last 8 bits specify writable register mask—only covers registers 16–23; rest of registers marked read-only

Allowed Operand Groups

<r1>$_{ag}$ = Gate Pointer$_{ag}$

Processor State: gfcache miss

GS ← **GS** + **4**
$\mathbf{GS.PC}_{tag}$ ← $\mathbf{PC}_{tag}$
GS.PC ← **PC+1**
$\mathbf{GS.A}_{tag}$ ← $\mathbf{A}_{tag}$
GS.A ← **A**
$\mathbf{GS.EP}_{tag}$ ← $\mathbf{EP}_{tag}$
GS.EP ← **EP**
$\mathbf{GS.R}_{ag}$ ← $\mathbf{Integer}_{ag}$
$\mathbf{GS.R}_{tag}$ ← *<r3>*$_{tag}$
GS.R ← (**WR**<<**32**) | **register_number**(*<r3>*) // *mixing tag, mask, and register in one atom to make compact*
$\mathbf{PC}_{tag}$ ← **TMU**
PC ← *<r1>*.**PC**
$\mathbf{A}_{tag}$ ← *<r1>*.$\mathbf{A}_{tag}$

A ← *<r1>*.**A**
EP$_{tag}$ ← *<r1>*.**EP**$_{tag}$
EP ← *<r1>*.**EP**
MR$_{tag}$ ← *<r1>*.**PC**$_{tag}$
MR$_{ag}$ ← *<r1>*.**PC**$_{ag}$
MR ← *<r1>*.**PC**
WR=**WR** & ((*[wmask]* & 0x03ff) <<16) // immediate 10b

Processor State: gfcache hit

PC ← **PC** + **1**
<r3> ← cached result of bgcall *<r1>* on *<r2>* and *<r3>*
<r3>$_{ag}$ ← atomic group from cached result of bgcall *<r1>* on *<r2>* and *<r3>*
<r3>$_{tag}$ ← *<r3>*$_{tag}$ // by formulation should stay the same
LR ← **LR** & $\overline{((\textit{[wmask]} \& 0x0ff) << 16)}$ // still mark writeable set dead – which are bottom 8b
LR[register_number(*<r3>*)]← 1

Error Conditions

<r1>$_{ag}$ ≠ Gate Pointer$_{ag}$
16 ≤ register_number(*<r3>*) < 24
<r3>[register_number(*<r3>*)-16] ≠ 1 // result register should be writeable
<r1>.**A** $_{ag}$≠ Authority$_{ag}$
<r1>.**PC**$_{ag}$ ≠ Instruction Pointer$_{ag}$
mask specified in *[mask]*[9:8] does not match mask read from gfcache

15.2.15 `bgacall <r1> <r2> <r3> [result-reg]` – Bracket call with authority change and augmentation

Invokes the bracket gate at *<r1>* with its authority extended by the first-class authority *<r2>* with the registers specified by *<r3>* as writeable, expecting the result to go in register *[result-reg]* which will have a tag *<r3>*$_{tag}$.

[Note: this combines result tag and write mask into a single atom (as it gets combined on the GS Stack), whereas bcall/bgcall do not. Necessary here to compact the arguments. Should we treat bcall/bgcall likewise? –AMD]

Operands

<r1> – Gate to call
<r2> – Additional Authority to use with gate
<r3> – bit vector specifying which registers will be writeable during the call; except for the designated return register, these will be cleared upon `brtn`; tagged with tag equal to tag for result
[result-reg] – specify register that should get result and be tagged by *<r2>*$_{tag}$

Allowed Operand Groups

<r1>$_{ag}$ = Gate Pointer$_{ag}$, *<r2>*$_{ag}$ = Authority$_{ag}$, *<r3>*$_{ag}$ = Integer$_{ag}$

Processor State

GS ← **GS** + **4**
GS.PC$_{tag}$ ← **PC**$_{tag}$
GS.PC ← **PC**+**1**

GS.A$_{tag}$ ← **A**$_{tag}$
GS.A ← **A**
GS.EP$_{tag}$ ← **EP**$_{tag}$
GS.EP ← **EP**
GS.R$_{ag}$ ← **Integer**$_{ag}$
GS.R$_{tag}$ ← *<r3>*$_{tag}$
GS.R ← **(WR<<32)**|*[result-reg]* // *mixing tag, mask, and register in one atom to make compact*
PC$_{tag}$ ← **TMU**
PC ← *<r1>*.**PC**
A ← *<r1>*.**A** with *<r2>* // from TMU output
EP$_{tag}$ ← *<r1>*.**EP**$_{tag}$
EP ← *<r1>*.**EP**
MR$_{tag}$ ← *<r1>*.**PC**$_{tag}$
MR$_{ag}$ ← *<r1>*.**PC**$_{ag}$
MR ← *<r1>*.**PC**
WR=**WR** & *<r3>*
// *see subtlety notes at end of gacall*

Error Conditions

<r1>$_{ag}$ ≠ Gate Pointer$_{ag}$
<r2>$_{ag}$ ≠ Authority$_{ag}$
<r3>$_{ag}$ ≠ Integer$_{ag}$
<r3>[*[result-reg]*] ≠ 1 // result register should be writeable
<r1>.**A** $_{ag}$≠ Authority$_{ag}$
<r1>.**PC**$_{ag}$ ≠ Instruction Pointer$_{ag}$

15.2.16 `brtn <r1>` – Return from Bracket Call

Return from a bracket call

Operands

<r1> – Source – return result register

Allowed Operand Groups

<r1>$_{ag}$ = Any$_{ag}$

Processor State

MR$_{ag}$ ← **GS.R**$_{ag}$
MR$_{tag}$ ← **GS.R**$_{tag}$ // *make available to TMU so can check that* ***PC***$_{tag}$ *and* *<r1>*$_{tag}$ *each flow to this*
MR ← **GS.R**
PC$_{tag}$ ← **GS.PC**$_{tag}$
PC ← **GS.PC**
A ← **GS.A**
EP$_{tag}$ ← **GS.EP**$_{tag}$
EP ← **GS.EP**
<r1>$_{tag}$ ← **GS.R**$_{tag}$
LR ← **LR** & $\overline{\textbf{WR}}$ // all writeable become dead
LR[register_number(*<r1>*)]← 1

WR ← **GS.R**>>**32**
GS ← **GS** - **4**

Error Conditions

GS.R$_{ag}$ ≠ Integer$_{ag}$
GS.R & 0x01f ≠ register_number(*<r1>*)

15.2.17 `tcall <r1> <r2>` – Procedure Call with timeout, no authority change

Call procedure in *<r1>* setting logical timeout to *<r2>*. Builds a return gate with the current authority and environment and pushes on gate stack.

Two things this does not protect against: (1) resource exhaustion, (2) stream deadlock. So, this cannot be a wholistic solution on its own to prevent a thread from never returning from a call.

Question remains: is this worthwhile to have, nonethless as part of our portfolio of protections? or should we just say all invocations of potentially suspicious code must be thread spawns and not bother with tcall/trtn/trequire?

Operands

<r1> – Pointer to procedure
<r2> – Number of instructions allowed to run

Allowed Operand Groups

<r1>$_{ag}$ = Instruction Pointer$_{ag}$, *<r2>*$_{ag}$ = Integer$_{ag}$

Processor State

GS ← **GS** + **4**
GS.PC$_{tag}$ ← **PC**$_{tag}$
GS.PC ← **PC+1**
GS.A$_{tag}$ ← **A**$_{tag}$
GS.A ← **A**
GS.EP$_{tag}$ ← **EP**$_{tag}$
GS.EP ← **EP**
GS.R$_{ag}$ ← **??**$_{ag}$ // distinguish from normal and bcall
GS.R ← **IL** // save old limit
GS.R$_{tag}$ ← **PC**$_{tag}$
PC$_{tag}$ ← **TMU**
EP← Uninitialized // *not coming form gate closure – **EP** is for the* lexically *enclosing environment, not the callers environment. So, not appropriate to chain through **EP**.*
IL← **IC**+*<r2>*
PC ← *<r1>*

Error Conditions

<r1>$_{ag}$ ≠ Instruction Pointer
<r2>$_{ag}$ ≠ Int
IC+*<r2>* > **IL** // should return appropriate NAV to enclosing brtn record

15.2.18 `trtn` – Return from Timeout Gate Call

Return from a timeout gate call

Operands

None

Allowed Operand Groups

None

Processor State

$\mathbf{PC}_{tag} \leftarrow \mathbf{TMU}$
$\mathbf{MR}_{ag} \leftarrow \mathbf{GS.PC}_{ag}$
$\mathbf{MR}_{tag} \leftarrow \mathbf{GS.PC}_{tag}$
$\mathbf{MR} \leftarrow \mathbf{GS.PC}$
$\mathbf{PC} \leftarrow \mathbf{GS.PC}$
$\mathbf{A} \leftarrow \mathbf{GS.A}$
$\mathbf{EP}_{tag} \leftarrow \mathbf{GS.EP}_{tag}$
$\mathbf{EP} \leftarrow \mathbf{GS.EP}$
$\mathbf{IL} \leftarrow \mathbf{GS.R}$
$\mathbf{GS} \leftarrow \mathbf{GS}$ - 4

Error Conditions

$\mathbf{GS.A}_{ag} \neq$ Authority
$\mathbf{GS.PC}_{ag} \neq$ Instruction Pointer
$\mathbf{GS.R}_{ag} =$ Integer$_{ag}$ or Error$_{ag}$ // will be Integer$_{ag}$ for bcall record and Error$_{ag}$ for gcall record

15.2.19 `trequire <r1>` – Require Time to Continue

Timeout now if *<r1>* logical cycles are not available before timeout.
Privilege: None

Operands

<r1> – Integer specifiying the time required

Allowed Operand Groups

<r1>$_{ag}$ = Integer$_{ag}$

Processor State

$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$

TMU Rule Match Fields

$\mathbf{PC}_{tag}$, $\mathbf{CI}_{tag}$, **CI**, **A**, *<r1>*$_{tag}$

Error Conditions

<r1>$_{ag} \neq$ Int
IC+*<r1>* > **IL** // should return appropriate NAV to enclosing brtn record

15.2.20 `gate <r1> <r2> <r3>` – Create Gate

Create a gate to procedure pointed to by *<r1>* with environment *<r2>* and store in pre-allocated gate structure *<r3>*

Operands

<r1> – Pointer to procedure
<r2> – Environment for gate
<r3> – Linear Pointer to previously allocated gate structure

Allowed Operand Groups

$\langle r1 \rangle_{ag}$ = Instruction Pointer$_{ag}$, $\langle r2 \rangle_{ag}$ = Frame Pointer$_{ag}$, $\langle r3 \rangle_{ag}$ = Linear Pointer$_{ag}$
maybe open type of *<r2>* up to all non-linear types? – at some point that was the original intent.

Processor State

$\mathbf{MR}_{tag} \leftarrow \mathbf{mem}[\langle r3 \rangle]_{tag}$ // so TMU can check [...but it cannot check all 3 atoms... –AMD]
$\mathbf{MR}_{ag} \leftarrow \mathbf{mem}[\langle r3 \rangle]_{ag}$
$\mathbf{MR} \leftarrow \mathbf{mem}[\langle r3 \rangle]$
$\langle r3 \rangle.\mathbf{PC}_{tag} \leftarrow \langle r1 \rangle_{tag}$
$\langle r3 \rangle.\mathbf{PC} \leftarrow \langle r1 \rangle$
$\langle r3 \rangle.\mathbf{A}_{tag} \leftarrow \mathbf{A}_{tag}$
$\langle r3 \rangle.\mathbf{A} \leftarrow \mathbf{A}$
$\langle r3 \rangle.\mathbf{EP}_{tag} \leftarrow \langle r2 \rangle_{tag}$
$\langle r3 \rangle.\mathbf{EP} \leftarrow \langle r2 \rangle$
$\langle r3 \rangle_{ag} \leftarrow \mathbf{GatePointer}_{ag}$
$\langle r3 \rangle_{tag} \leftarrow \mathbf{TMU}$

Error Conditions

<r3> not point to base of frame [these three possible additions –AMD]
<r3> size incorrect for gate frame ??
Contents of memory at *<r3>* not Uninitialized

15.2.21 `gatele <r1> <r2> <r3>` – Create Gate with Linear Environment

Create a gate to procedure pointed to by *<r1>* with environment *<r2>* and store in pre-allocated gate structure *<r3>*. Move *<r2>* into the gate.

Operands

<r1> – Pointer to procedure
<r2> – Environment for gate
<r3> – Linear Pointer to previously allocated gate structure

Allowed Operand Groups

$\langle r1 \rangle_{ag}$ = Instruction Pointer$_{ag}$, $\langle r2 \rangle_{ag}$ = Frame Pointer$_{ag}$, $\langle r3 \rangle_{ag}$ = Linear Pointer$_{ag}$
$\langle r1 \rangle_{ag}$ = Instruction Pointer$_{ag}$, $\langle r2 \rangle_{ag}$ = Stream Write Pointer$_{ag}$, $\langle r3 \rangle_{ag}$ = Linear Pointer$_{ag}$
$\langle r1 \rangle_{ag}$ = Instruction Pointer$_{ag}$, $\langle r2 \rangle_{ag}$ = Stream Read Pointer$_{ag}$, $\langle r3 \rangle_{ag}$ = Linear Pointer$_{ag}$

<r1>$_{ag}$ = Instruction Pointer$_{ag}$, *<r2>*$_{ag}$ = Linear Pointer$_{ag}$, *<r3>*$_{ag}$ = Linear Pointer$_{ag}$ [if keep linear atomic group –AMD]
maybe open *<r2>* to any type?

Processor State

MR$_{tag}$ ← **mem**[*<r3>*]$_{tag}$ // so TMU can check [...but it cannot check all 3 atoms... –AMD]
MR$_{ag}$ ← **mem**[*<r3>*]$_{ag}$
MR ← **mem**[*<r3>*]
<r3>.**PC**$_{tag}$ ← *<r1>*$_{tag}$
<r3>.**PC** ← *<r1>*
<r3>.**A**$_{tag}$ ← **A**$_{tag}$
<r3>.**A** ← **A**
<r3>.**EP**$_{tag}$ ← *<r2>*$_{tag}$
<r3>.**EP** ← *<r2>*
<r3>$_{ag}$ ← **GatePointer**$_{ag}$
<r3>$_{tag}$ ← **TMU**
<r2>$_{tag}$ ← **TMU.2**
<r2> ← **0**

Error Conditions

<r3> not point to base of frame [these three possible additions –AMD]
<r3> size incorrect for gate frame ??
Contents of memory at *<r3>* not Uninitialized

15.3 Timer

15.3.1 `give-time <r1>` – Add time to SLOT

Add the time specified in *<r1>* to **SLOT**.
Privilege: Only BTSP is allowed to execute this instruction.

Operands

<r1> – Integer specifiying the time to allocate.

Allowed Operand Groups

<r1>$_{ag}$ = Integer$_{ag}$

Processor State

PC ← **PC** + **1**
SLOT ← **SLOT** + *<r1>*

TMU Rule Match Fields

PC$_{tag}$, **CI**$_{tag}$, **CI**, **A**, *<r1>*$_{tag}$

15.3.2 `recover-time <r1>` – Move time remaining from SLOT to *<r1>*

Set **SLOT** to 1 and put its old value in *<r1>*.

Note: we set it to 1 rather than 0 to guarantee there is always a full QT left after this operation.

Privilege: Only BTSP is allowed to execute this instruction.

Operands

<r1> – register to hold result.

Processor State

PC ← **PC** + **1**
if (SLOT>0) *<r1>* ← **SLOT**-1 else *<r1>* ← 0
if (SLOT>0) **SLOT** ← 1 else **SLOT** ← 0

TMU Rule Match Fields

$\mathbf{PC}_{tag}$, $\mathbf{CI}_{tag}$, **CI**, **A**, $\textit{<r1>}_{tag}$

15.3.3 `read-time <r1>` – Read time remaining in SLOT to *<r1>*

Set *<r1>* to **SLOT**.

Privilege: Only schedulers are allowed to perform this operation.

Operands

<r1> – register to hold result.

Processor State

PC ← **PC** + **1**
<r1> ← **SLOT**

TMU Rule Match Fields

$\mathbf{PC}_{tag}$, $\mathbf{CI}_{tag}$, **CI**, **A**, $\textit{<r1>}_{tag}$

15.4 Inter-thread Control Flow

15.4.1 `runt <r1>` – Run Thread

Begin running a new thread by storing **TP** into **TTT** and replacing **TP** with *<r1>*.

Privilege: Only BTSP is allowed to execute this instruction.

Operands

<r1> – Thread pointer to begin running

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = Thread Pointer_{ag}

Processor State

PC ← **PC** + **1**
TTT ← **TP**
TP ← *<r1>*

Assigning **TP** causes all processor state to be replaced with the new thread state.

15.4.2 `resumet <r1> [event]` – Return to Thread from Trap Handler

Begin running a new thread by storing **TP** into **r1**.ITT[*[event]*] and replacing **TP** with *<r1>*.
Privilege: Limited to fault handlers

Operands

<r1> – Thread pointer to begin running
[event] – slot in ITT where the current **TP** should be returned

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = Thread Pointer$_{ag}$
$\textit{<r2>}_{ag}$ = Integer$_{ag}$

Processor State

PC ← **PC + 1**
<r1>.ITT[*[event]*] ← **TP**
<r1>.**TS** .ErrorCode ← No Error
TP ← *<r1>*

Assigning **TP** causes all processor state to be replaced with the new thread state.

15.4.3 `yield` – Yield remaining time

Invokes a timer interrupt without further operations.
Privilege: none

Operands

none

Processor State

PC ← **PC + 1**
SLOT ← **SLOT**+1
TP ← **TTT**
TTT ← Empty
TS.ThreadState ← Voluntary Yield
perform timer interrupt synchronously **after** this instruction and **before** the following instruction

Note: this doesn't add time; it just merges the remaining time into the normal scheduling time slot that occurs after a timer interrupt.

TMU Rule Match Fields

PC$_{tag}$, **CI**$_{tag}$, **CI**, **A**

15.4.4 `yield2 <r1> [event]` – Return to TTT Bypassing Stacked Thread

Begin running a new thread by storing **TP** into **r1**.ITT[*[event]*] and replacing **TP** with **TTT**.
Privilege: Limited to fault handlers

Operands

<r1> – Thread pointer into which to store **TP**
[event] – slot in ITT where the current **TP** should be returned

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = Thread Pointer$_{ag}$
$\textit{<r2>}_{ag}$ = Integer$_{ag}$

Processor State

PC ← **PC** + **1**
<r1>.ITT[*[event]*] ← **TP**
TTT.**EP** ← *<r1>*
<r1> ← Empty
TP ← **TTT**
TTT ← Empty

15.4.5 `endt` – Thread self termination

Change thread state to halted and invoke time interrupt without further operations.
Privilege: none

Operands

none

Processor State

PC ← **PC** + **1**
SLOT ← **SLOT**+1
TP ← **TTT**
TS.ThreadState ← Halted
perform timer interrupt synchronously **after** this instruction and **before** the following instruction

Note: this doesn't add time; it just merges the remaining time into the normal scheduling time slot that occurs after a timer interrupt.

TMU Rule Match Fields

PC$_{tag}$, **CI**$_{tag}$, **CI**, **A**

Assigning **TP** causes all processor state to be replaced with the new thread state.

16 Memory OP Codes

Memory op-codes use fat pointers to refer to atoms in any frame the code has a capability for.

16.1 Memory Access

16.1.1 `clear <r1>` – Clear Register

Clear *<r1>*, marking it as dead. The old value of *<r1>* can no longer be read.

Operands

<r1> – Destination – register to clear

Allowed Operand Groups

$\textit{<r1>}_{ag} = \text{Any}_{ag}$

Processor State

PC $\leftarrow$ **PC** + **1**
$\textit{<r1>}_{tag} \leftarrow$ **Public**
<r1> $\leftarrow 0$
$\textit{<r1>}_{ag} \leftarrow \textbf{Error}_{ag}$
LR[register_number(*<r1>*)] $\leftarrow$ false

16.1.2 `clearregs <r1>` – Clear Group of Registers

Clear registers specified by *<r1>*, marking them as dead. The old value of this register set can no longer be read.

Operands

<r1> – Source 1 – registers to clear

Allowed Operand Groups

$\textit{<r1>}_{ag} = \text{Integer}_{ag}$

Processor State

PC $\leftarrow$ **PC** + **1**
LR $\leftarrow$ **LR** & $\overline{\textit{<r1>}}$

16.1.3 `livemask <r1>` – Read the Live Register mask

Put the Live Register mask into *<r1>*

Operands

<r1> – Destination

Allowed Operand Groups

$\textit{<r1>}_{ag} = \text{Any}_{ag}$

Processor State

PC $\leftarrow$ **PC** + **1**
$\textit{<r1>}_{tag} \leftarrow$ **TMU**
$\textit{<r1>}_{ag} \leftarrow \textbf{Integer}_{ag}$
<r1> $\leftarrow$ **LR**

16.1.4 `writemask <r1>` – Read the Writeable Register mask

Put the Writeable Register mask into *<r1>*

Operands

<r1> – Destination

Allowed Operand Groups

$\textit{<r1>}_{ag} = \text{Any}_{ag}$

Processor State

PC ← **PC + 1**
$\textit{<r1>}_{tag}$ ← **TMU**
$\textit{<r1>}_{ag}$ ← $\textbf{Integer}_{ag}$
<r1> ← **WR**

16.1.5 `mvrr <r1> <r2>` – Move Register To Register

Move an atom from *<r1>* into *<r2>*, erasing *<r1>*

Operands

<r1> – Frame Pointer register
<r2> – Destination register

Allowed Operand Groups

all

Processor State

PC ← **PC + 1**
$\textit{<r2>}_{tag}$ ← **TMU**
<r2> ← *<r1>*
$\textit{<r2>}_{ag}$ ← $\textit{<r1>}_{ag}$
$\textit{<r1>}_{tag}$ ← **TMU.2**
<r1> ← **0**
$\textit{<r1>}_{ag}$ ← $\textbf{Error}_{ag}$

16.1.6 `mvmr <r1> <r2>` – Move Memory To Register

Load an atom from *<r1>* into *<r2>*, erasing memory location at *<r1>*

Operands

<r1> – Pointer register
<r2> – Destination register

Allowed Operand Groups

<r1>$_{ag}$ = Frame Pointer$_{ag}$
<r1>$_{ag}$ = Constant Frame Pointer$_{ag}$
<r1>$_{ag}$ = Instruction Pointer$_{ag}$
<r1>$_{ag}$ = Thread Pointer$_{ag}$
<r1>$_{ag}$ = Linear Pointer$_{ag}$

Processor State

PC $\leftarrow$ **PC + 1**
MR $\leftarrow$ **mem**[*<r1>*]
MR$_{tag}$ $\leftarrow$ **mem**[*<r1>*]$_{tag}$
MR$_{ag}$ $\leftarrow$ **mem**[*<r1>*]$_{ag}$
<r2>$_{tag}$ $\leftarrow$ **TMU**
<r2> $\leftarrow$ **mem**[*<r1>*]
<r2>$_{ag}$ $\leftarrow$ **mem**[*<r1>*]$_{ag}$
mem[*<r1>*]$_{tag}$ $\leftarrow$ **TMU**.2
mem[*<r1>*] $\leftarrow$ **0**
mem[*<r1>*]$_{ag}$ $\leftarrow$ **Error**$_{ag}$

16.1.7 `mvrm <r1> <r2>` – Move Register To Memory

Store an atom from *<r1>* into *<r2>*, erasing *<r1>*.

Operands

<r1> – Source register
<r2> – Pointer register

Allowed Operand Groups

<r2>$_{ag}$ = Frame Pointer$_{ag}$
<r2>$_{ag}$ = Instruction Pointer$_{ag}$
<r2>$_{ag}$ = Thread Pointer$_{ag}$
<r2>$_{ag}$ = Linear Pointer$_{ag}$

Processor State

PC $\leftarrow$ **PC + 1**
MR $\leftarrow$ **mem**[*<r2>*]
MR$_{tag}$ $\leftarrow$ **mem**[*<r2>*]$_{tag}$
MR$_{ag}$ $\leftarrow$ **mem**[*<r2>*]$_{ag}$
mem[*<r2>*]$_{tag}$ $\leftarrow$ **TMU**
mem[*<r2>*] $\leftarrow$ *<r1>*
mem[*<r2>*]$_{ag}$ $\leftarrow$ *<r1>*$_{ag}$
<r1>$_{tag}$ $\leftarrow$ **TMU.2**
<r1> $\leftarrow$ **0**
<r1>$_{ag}$ $\leftarrow$ **Error**$_{ag}$

16.1.8 `cprr <r1> <r2>` – Copy Register To Register

Copy an atom from *<r1>* into *<r2>*.

Operands

<r1> – Source register
<r2> – Destination register

Allowed Operand Groups

$\textit{<r1>}_{ag}$ all but Thread Pointer, Linear Pointer, or Stream Pointer

Processor State

$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$
$\textit{<r2>}_{tag} \leftarrow \mathbf{TMU}$
$\textit{<r2>} \leftarrow \textit{<r1>}$
$\textit{<r2>}_{ag} \leftarrow \textit{<r1>}_{ag}$

16.1.9 `cpmr <r1> <r2>` – Copy Memory To Register

Load an atom from memory at address *<r1>* into *<r2>*.

Operands

<r1> – Pointer register
<r2> – Destination register

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = Frame Pointer$_{ag}$
$\textit{<r1>}_{ag}$ = Constant Frame Pointer$_{ag}$
$\textit{<r1>}_{ag}$ = Instruction Pointer$_{ag}$
$\textit{<r1>}_{ag}$ = Thread Pointer$_{ag}$
$\textit{<r1>}_{ag}$ = Linear Pointer$_{ag}$

Error Conditions

Value read from memory must not be a linear pointer

Processor State

$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$
$\mathbf{MR} \leftarrow \mathbf{mem}[\textit{<r1>}]$
$\mathbf{MR}_{tag} \leftarrow \mathbf{mem}[\textit{<r1>}]_{tag}$
$\mathbf{MR}_{ag} \leftarrow \mathbf{mem}[\textit{<r1>}]_{ag}$
$\textit{<r2>}_{tag} \leftarrow \mathbf{TMU}$
$\textit{<r2>} \leftarrow \mathbf{mem}[\textit{<r1>}]$
$\textit{<r2>}_{ag} \leftarrow \mathbf{mem}[\textit{<r1>}]_{ag}$

16.1.10 `cprm <r1> <r2>` – Copy Register To Memory

Store an atom from *<r1>* into memory address *<r2>*.

Operands

<r1> – Source register
<r2> – Pointer register

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = all but Thread Pointer, Linear Pointer, or Stream Pointer$_{ag}$
$\textit{<r2>}_{ag}$ = Frame Pointer$_{ag}$
$\textit{<r2>}_{ag}$ = Instruction Pointer$_{ag}$
$\textit{<r2>}_{ag}$ = Thread Pointer$_{ag}$
$\textit{<r2>}_{ag}$ = Linear Pointer$_{ag}$

Processor State

$\mathbf{PC} \leftarrow \mathbf{PC} + 1$
$\mathbf{MR} \leftarrow \mathbf{mem}[\textit{<r2>}]$
$\mathbf{MR}_{tag} \leftarrow \mathbf{mem}[\textit{<r2>}]_{tag}$
$\mathbf{MR}_{ag} \leftarrow \mathbf{mem}[\textit{<r2>}]_{ag}$
$\mathbf{mem}[\textit{<r2>}]_{tag} \leftarrow \mathbf{TMU}$
$\mathbf{mem}[\textit{<r2>}] \leftarrow \textit{<r1>}$
$\mathbf{mem}[\textit{<r2>}]_{ag} \leftarrow \textit{<r1>}_{ag}$

16.2 Pointers

Memory frames are created by subdivision of parent frames into child frames. At boot time the entirety of memory is contained in one master frame which is subsequently divided into sub-regions.

16.2.1 `lcfp <r1> [offset]` – Load Constant Frame Pointer

Creates a Constant Frame Pointer pointing to the address $\mathbf{PC}_{base}$ + *[offset]* and puts the pointer into *<r1>*

Operands

<r1> – Destination register
[offset] – Unsigned offset in current instruction frame

Allowed Operand Groups

None

Processor State

$\mathbf{PC} \leftarrow \mathbf{PC} + 1$
$\textit{<r1>}_{tag} \leftarrow \mathbf{TMU}$
<r1> ← (A=**PC**.Base+*[offset]*,Base=**PC**.Base,Len=**PC**.Len,update Finger accordingly)
$\textit{<r1>}_{ag} \leftarrow \mathbf{Constant\ Frame\ Pointer}_{ag}$

16.2.2 `framptr <r1> <r2> <r3>` – Create pointer to frame

Creates a pointer to a sub-region within a parent frame pointed to by *<r2>*. The size of the sub-region is specified as an L, B pair in *<r1>*. The newly created pointer is put in *<r3>*. Register *<r2>* remains unchanged.
Privilege: Only allocators are allowed to execute this instruction.

Operands

<r1> – Size of sub-region as an L, B pair
<r2> – Pointer to parent frame
<r3> – New pointer to sub-region frame

Allowed Operand Groups

$\langle r1 \rangle_{ag}$ = Integer$_{ag}$, $\langle r2 \rangle_{ag}$ = Frame Pointer$_{ag}$

Processor State

PC $\leftarrow$ **PC + 1**
$\langle r3 \rangle_{tag} \leftarrow$ **TMU**
<r3> $\leftarrow$ (A=r2.A, r1.L, r1.B, F=0)
$\langle r3 \rangle_{ag} \leftarrow \langle r2 \rangle_{ag}$

Error Conditions

The pointer in *<r2>* does not have the correct alignment for the size specified in *<r1>*
The pointer in *<r2>* does not have adequate space for the requested allocation

16.2.3 `offp <r1> <r2> <r3>` – Offset Pointer

Add the Integer$_{ag}$ atom in *<r1>* to pointer atom *<r2>*, result in *<r3>*

Operands

<r1> – Offset
<r2> – Pointer
<r3> – Destination

Allowed Operand Groups

$\langle r1 \rangle_{ag}$ = Integer$_{ag}$, $\langle r2 \rangle_{ag}$ = FramePointer$_{ag}$
$\langle r1 \rangle_{ag}$ = Integer$_{ag}$, $\langle r2 \rangle_{ag}$ = InstructionPointer$_{ag}$

(it is intentional that this does not operate on ConstantFramePointer)

Processor State

PC $\leftarrow$ **PC + 1**
$\langle r3 \rangle_{tag} \leftarrow$ **TMU**
<r3> $\leftarrow$ *<r1>* + *<r2>*
$\langle r3 \rangle_{ag} \leftarrow \langle r2 \rangle_{ag}$

16.2.4 `offlp <r1> <r2>` – Offset Linear Pointer

Add the Integer$_{ag}$ atom in *<r1>* to pointer atom *<r2>*, overwriting *<r2>* with result

Operands

<r1> – Offset
<r2> – Pointer

Allowed Operand Groups

<r1>$_{ag}$ = Integer$_{ag}$, *<r2>*$_{ag}$ = FramePointer$_{ag}$
<r1>$_{ag}$ = Integer$_{ag}$, *<r2>*$_{ag}$ = InstructionPointer$_{ag}$
<r1>$_{ag}$ = Integer$_{ag}$, *<r2>*$_{ag}$ = LinearFramePointer$_{ag}$
<r1>$_{ag}$ = Integer$_{ag}$, *<r2>*$_{ag}$ = ThreadPointer$_{ag}$

(it is intentional that this does not operate on ConstantFramePointer)

Processor State

PC ← **PC + 1**
<r2>$_{tag}$ ← **TMU**
<r2> ← *<r1>* + *<r2>*
<r2>$_{ag}$ ← *<r2>*$_{ag}$

16.2.5 `offtp <r1> <r2>` – Offset Thread Pointer

Add the Integer$_{ag}$ atom in *<r1>* to pointer atom *<r2>*, overwriting *<r2>* with result. Similar to offlp, but (i) limited to Miss Handler, (ii) not look at *<r2>* tag, (iii) perserves *<r2>* tag, (iv) only works on Thread Pointers.
Privilege: Limited to TMUMissHandle

Operands

<r1> – Offset
<r2> – Pointer

Allowed Operand Groups

<r1>$_{ag}$ = Integer$_{ag}$, *<r2>*$_{ag}$ = ThreadPointer$_{ag}$

Processor State

PC ← **PC + 1**
<r2>$_{tag}$ ← *<r2>*$_{tag}$ // identity
<r2> ← *<r1>* + *<r2>*
<r2>$_{ag}$ ← *<r2>*$_{ag}$

16.2.6 `basep <r1> <r2>` – Pointer Base

Put the base pointer for the pointer in *<r1>* into *<r2>*

Operands

<r1> – Pointer
<r2> – Destination

Allowed Operand Groups

$\langle r1 \rangle_{ag}$ = FramePointer$_{ag}$
$\langle r1 \rangle_{ag}$ = InstructionPointer$_{ag}$
$\langle r1 \rangle_{ag}$ = ConstantFramePointer$_{ag}$ // use on ConstantFramePointer might be privileged even if use on FramePointer is not
$\langle r1 \rangle_{ag}$ = Integer$_{ag}$, $\langle r2 \rangle_{ag}$ = ThreadPointer$_{ag}$ // privileged
$\langle r1 \rangle_{ag}$ = Integer$_{ag}$, $\langle r2 \rangle_{ag}$ = GatePointer$_{ag}$ //prividged

Processor State

PC ← **PC** + **1**
$\langle r2 \rangle_{tag}$ ← **TMU**
<r2> ← base(*<r1>*) // how depends on fat-pointer encoding
$\langle r2 \rangle_{ag}$ ← $\langle r1 \rangle_{ag}$

16.2.7 `sizep <r1> <r2>` – Pointer Size

Put the size (as an L, B pair) for the pointer in *<r1>* into *<r2>*
Privilege: Limited to appropriate piece of GC
[Maybe not limit? If one can increment a pointer until get a NAV (and the NAV is classified the same as the pointer, then it should be possible to get at this information. So, if that's safe, it should be safe to ask the size of the frame a pointer points to. –AMD]

Operands

<r1> – Pointer
<r2> – Destination as an L, B pair [I suspect the primary use is to turn around and use in an allocation, but might merit some discussion. –AMD]

Allowed Operand Groups

$\langle r1 \rangle_{ag}$ = FramePointer$_{ag}$
$\langle r1 \rangle_{ag}$ = InstructionPointer$_{ag}$
$\langle r1 \rangle_{ag}$ = ConstantFramePointer$_{ag}$
$\langle r1 \rangle_{ag}$ = Integer$_{ag}$, $\langle r2 \rangle_{ag}$ = ThreadPointer$_{ag}$ // privileged
$\langle r1 \rangle_{ag}$ = Integer$_{ag}$, $\langle r2 \rangle_{ag}$ = GatePointer$_{ag}$ // privileged

Processor State

PC ← **PC** + **1**
$\langle r2 \rangle_{tag}$ ← **TMU**
<r2> ← size(*<r1>*) // how depends on fat-pointer encoding
$\langle r2 \rangle_{ag}$ ← Integer$_{ag}$

16.2.8 `fphash <r1> <r2>` – Hash Frame Pointer

Create a hash in *<r2>* of the frame pointer in *<r1>*.
Privilege: Likely limited

Operands

<r1> – Frame Pointer to Hash
<r2> – Integer representing the hash of *<r1>*

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = Frame Pointer$_{ag}$

Processor State

PC ← **PC** + **1**
$\textit{<r2>}_{tag}$ ← **TMU**
<r2> ← hash(*<r1>*)
$\textit{<r2>}_{ag}$ ← Integer$_{ag}$

17 Security OP Codes

17.1 Authority Management

17.1.1 `seta <r1>` – Set Authority

Set Principal to *<r1>*
Privilege: very privileged.

Operands

<r1> – authority to load $\textit{<r1>}_{ag}$=Authority

Processor State

PC ← **PC** + **1**
$\mathbf{A}_{tag}$ ← $\textit{<r1>}_{tag}$
A ← *<r1>*

17.1.2 `raisea <r1>` – Augment current authority

Extend current authority by first-class authority in *<r1>*.

Operands

<r1> – Additional Authority to add to current authority

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = Authority$_{ag}$

Processor State

PC ← **PC** + **1**
A ← **TMU** // which should be **A** with *<r1>*

17.1.3 `lowera <r1>` – Refine current authority

Remove first-class authority in *<r1>* from current authority.

Operands

<r1> – Authority to remove from current authority

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = Authority_{ag}

Processor State

PC ← **PC** + **1**
A ← **TMU** // which should be **A** without *<r1>*

17.1.4 `ina <r1> <r2>` – Inspect Authority

Convert Authority in *<r1>* to frame-pointer in *<r2>*.
Privilege: Limited to AuthModel

Operands

<r1> – Authority to inspect
<r2> – Frame pointer associated with authority

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = Authority_{ag}

Processor State

PC ← **PC** + **1**
$\textit{<r2>}_{tag}$ ← **TMU**
<r2> ← *<r1>*
$\textit{<r2>}_{ag}$ ← FramePtr_{ag}

17.1.5 `inp <r1> <r2>` – Inspect Principal

Convert Principal in *<r1>* to frame-pointer in *<r2>*.
Privilege: Limited to AuthModel

Operands

<r1> – Principal to inspect
<r2> – Frame pointer associated with principal

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = Principal_{ag}

Processor State

PC ← **PC** + **1**
$\textit{<r2>}_{tag}$ ← **TMU**
<r2> ← *<r1>*
$\textit{<r2>}_{ag}$ ← FramePtr_{ag}

17.1.6 `cpar <r1> <r2>` – Read Authority from Memory

Put the authority from the memory location pointed to by *<r1>* into *<r2>*. Similar to cpmr, but rules coded not to care about $\texttt{<r1>}_{tag}$; can only be used to read authorities.
Privilege: Limited to TMUMissHandle

Operands

<r1> – Pointer to memory location holding an authority
<r2> – Register to hold the authority

Allowed Operand Groups

$\texttt{<r1>}_{ag}$ = Thread Pointer$_{ag}$, $\texttt{<MR>}_{ag}$ = Authority$_{ag}$

Processor State

PC ← **PC + 1**
MR ← **mem**[*<r1>*]
$\mathbf{MR}_{tag}$ ← **mem**[*<r1>*]$_{tag}$ // this gets ignored
$\mathbf{MR}_{ag}$ ← **mem**[*<r1>*]$_{ag}$
$\texttt{<r2>}_{tag}$ ← **TMU**
$\texttt{<r2>}_{ag}$ ← **Authority**$_{ag}$
<r2> ← **MR**

17.1.7 `cpio <r1> <r2>` – Read Opcode from Instruction in Memory

Put the opcode for the instruction in the memory location pointed to by *<r1>* into *<r2>*.
Privilege: Limited to TMUMissHandle

Operands

<r1> – Pointer to memory location whose tag is to be extracted
<r2> – Integer for extracted OpCode

Allowed Operand Groups

$\texttt{<r1>}_{ag}$ = Thread Pointer$_{ag}$, $\texttt{<mem[<r1>]>}_{ag}$ = Instruction$_{ag}$

Processor State

PC ← **PC + 1**
MR ← **mem**[*<r1>*]
$\mathbf{MR}_{tag}$ ← **mem**[*<r1>*]$_{tag}$ // don't really care about this
$\mathbf{MR}_{ag}$ ← **mem**[*<r1>*]$_{ag}$ // need to check an Insruction
$\texttt{<r2>}_{tag}$ ← **TMU**
$\texttt{<r2>}_{ag}$ ← **Integer**$_{ag}$
<r2> ← (**MR** >> OP_CODE_LOCATION) && OP_CODE_MASK

17.2 TMU Management

17.2.1 `tmul <r1> <r2> <r3>` – TMU Load

Load atom in *<r3>* into field *<r2>* for rule *<r1>*.
Privilege: only the appropriate piece of the TMU handler can perform this operation

Operands

<r1> – TMU slot for rule to load
<r3> – Field within rule to load
<r3> – Value to load into field

Allowed Operand Groups

$\langle r1 \rangle_{ag}$ = Integer$_{ag}$, $\langle r2 \rangle_{ag}$ = Integer$_{ag}$, $\langle r3 \rangle_{ag}$ = Tag$_{ag}$ or $\langle r3 \rangle_{ag}$ = FramePointer$_{ag}$ or $\langle r3 \rangle_{ag}$ = Authority$_{ag}$ or $\langle r3 \rangle_{ag}$ = Integer$_{ag}$

Processor State

PC ← PC + 1
TMU[*<r1>*][*<r2>*] ← *<r3>*

TMU Load operations work within transactions, with the rule changes being committed to the TMU only on a successfully executed `transend`.

17.2.2 `tmuu <r1>` – TMU Unload

Unload rule in TMU slot *<r1>* and clear associated hit counter.
Privilege: only the appropriate piece of the TMU handler can perform this operation

Operands

<r1> – TMU slot to unload

Allowed Operand Groups

$\langle r1 \rangle_{ag}$ = Integer$_{ag}$

Processor State

PC ← PC + 1
TMU[*<r1>*][*]=Don't Care
TMUHitCount[*<r1>*]← 0

TMU unload operations work within transactions, with the rule changes being committed to the TMU only on a successfully executed `transend`.

17.2.3 `tmurc <r1> <r2>` – TMU Read Hit Counter

Read hit count for TMU Rule loaded into slot *<r1>* into register *<r2>*.
Privilege: only the appropriate piece of the TMU handler can perform this operation

Operands

<r1> – TMU slot's hit count to query
<r2> – register to hold hit count

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = Integer_{ag}

Processor State

PC ← **PC + 1**
<r2> ← TMUHitCount[*<r1>*]

17.2.4 `cphr <r1> <r2>` – Read Hash from Memory

Put the hash from the memory location pointed to by *<r1>* into *<r2>*. Similar to cpmr, but rules coded not to care about $\textit{<r1>}_{tag}$; can only be used to read integers.
Privilege: Limited to TMUMissHandle

Operands

<r1> – Pointer to memory location holding hash
<r2> – Register to hold hash once read

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = $\text{Thread Pointer}_{ag}$, $\textit{<MR>}_{ag}$ = Integer_{ag}

Processor State

PC ← **PC + 1**
MR ← **mem**[*<r1>*]
$\mathbf{MR}_{tag}$ ← $\mathbf{mem}[\textit{<r1>}]_{tag}$ // this gets ignored
$\mathbf{MR}_{ag}$ ← $\mathbf{mem}[\textit{<r1>}]_{ag}$
$\textit{<r2>}_{tag}$ ← **TMU**
$\textit{<r2>}_{ag}$ ← $\mathbf{Integer}_{ag}$
<r2> ← **MR**

17.2.5 `gfwrite <r1> <r2> <r3>` – gfcache write

Load atom in *<r1>* into slot *<r2>* of rule *<r3>*.
Privilege: Limited to GFA

Operands

<r1> – Part of rule to load (this is the data to be loaded)
<r2> – Slot (field) in rule to load (0 – result value; 1 – allowed mask)

<r3> – Llocation to load

Allowed Operand Groups

$\textit{<r1>}_{ag} = \text{Integer}_{ag}$, $\textit{<r2>}_{ag} = \text{Integer}_{ag}$, $\textit{<r3>}_{ag} = \text{Integer}_{ag}$

Processor State

PC ← **PC** + **1**
gfcache[*<r3>*][*<r2>*] ← *<r1>*

17.3 Tag Management

17.3.1 `newt <r1> <r2>` – New Tag

Create a tag from data in *<r1>* and applies it to *<r2>* leaving result in *<r2>*. This instruction overrides the **TMU** result.
Privilege: [if this is does create a tag from an allocated memory, then this is likely limited to the tag allocator –AMD]

Operands

<r1> – Source to convert to tag
<r2> – Destination with tag applied

Processor State

PC ← **PC** + **1**
$\textit{<r2>}_{tag}$ ← *<r1>*

17.3.2 `intag <r1> <r2>` – Inspect Tag

Convert tag on *<r1>* to integer in *<r2>*. [(Maybe frame pointer – think gets replaced by `int` which produces a pointer) –AMD]
Privilege: certainly limited. Parts of extern path?

Operands

<r1> – Tag to inspect, data ignored
<r2> – Result data is tag value

Processor State

PC ← **PC** + **1**
$\textit{<r2>}_{tag}$ ← **TMU**
<r2> ← $\textit{<r1>}_{tag}$
$\textit{<r2>}_{ag}$ ← Integer_{ag} [Maybe Frame Pointer –AMD]

[May need an analogous inp. –AMD]

17.3.3 `retag <r1> <r2>` – Retag Data

Apply tag on *<r1>* to data in *<r2>* and put combined result into *<r2>*. TMU rules specify which authorities can perform this retagging for specific tags.

Operands

<r1> – Tag to apply, data ignored
<r2> – Result data with tag applied

Processor State

$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$
$\langle r2 \rangle_{tag} \leftarrow \langle r1 \rangle_{tag}$

17.3.4 `settag <r1> <r2>` – Retag Data blind to current tags

Apply tag on *<r1>* to data in *<r2>* and put combined result into *<r2>*. This is the same operation as `retag`, except it allows different TMU rules. In particular, this one only looks at the authority executing the instruction.
Privilege: Limited to TMUMissHandle

Operands

<r1> – Tag to apply, data ignored
<r2> – Result data with tag applied

Processor State

$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$
$\langle r2 \rangle_{tag} \leftarrow \langle r1 \rangle_{tag}$

17.3.5 `retagpc <r1>` – Retag Program Counter

Apply tag on *<r1>* to **PC**. TMU rules specify which authorities can perform this retagging for specific tags.

Operands

<r1> – Tag to apply, data ignored

Processor State

$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$
$\mathbf{PC}_{tag} \leftarrow \langle r1 \rangle_{tag}$

17.3.6 `tagof <r1> <r2>` – Extract First-Class Tag

Put a first-class tag representation of the tag on *<r1>* into *<r2>*.

Operands

<r1> – Atom whose tag is to be extracted
<r2> – First-class tag representation of extracted tag

Processor State

PC ← **PC** + **1**
<r2>$_{tag}$ ← *<r1>*$_{tag}$
<r2>$_{ag}$ ← Tag$_{ag}$
<r2> ← (A=*<r1>*$_{tag}$.A, L=*<r1>*$_{tag}$.L, B=*<r1>*$_{tag}$.B, F=0)

17.3.7 `tagofpc <r1>` – Extract First-Class Tag for PC

Put a first-class tag representation of the tag on **PC** into *<r1>*.

Operands

<r1> – First-class tag representation of extracted tag

Processor State

PC ← **PC** + **1**
<r1>$_{tag}$ ← **PC**$_{tag}$
<r1>$_{ag}$ ← Tag$_{ag}$
<r1> ← (A=**PC**$_{tag}$.A, L=**PC**$_{tag}$.L, B=**PC**$_{tag}$.B, F=0)

17.3.8 `rflct <r1> <r2>` – Extract Pointer for Tag

Put the pointer for the tag on *<r1>* into *<r2>*.
Privilege: Limited to TMUMissHandle – possibly now only used for getting **TP**$_{tag}$

Operands

<r1> – Atom whose tag is to be extracted
<r2> – Pointer for the extracted tag

Processor State

PC ← **PC** + **1**
<r2>$_{tag}$ ← **TMU**
<r2>$_{ag}$ ← FramePointer$_{ag}$
<r2> ← (A=*<r1>*$_{tag}$.A, L=*<r1>*$_{tag}$.L, B=*<r1>*$_{tag}$.B, F=0)

17.3.9 `cpmt <r1> <r2>` – Read Pointer from Tag in Memory

Put the pointer for the tag on the memory location pointed to by *<r1>* into *<r2>*.
Privilege: Limited to TMUMissHandle

Operands

<r1> – Pointer to memory location whose tag is to be extracted
<r2> – Pointer for the extracted tag

Allowed Operand Groups

<r1>$_{ag}$ = Thread Pointer$_{ag}$

Processor State

PC ← **PC** + **1**
MR ← **mem**[*<r1>*] // don't really care about this
$\mathbf{MR}_{tag}$ ← $\mathbf{mem}[\textit{<r1>}]_{tag}$
$\mathbf{MR}_{ag}$ ← $\mathbf{mem}[\textit{<r1>}]_{ag}$ // don't really care about this
$\textit{<r2>}_{tag}$ ← **TMU**
$\textit{<r2>}_{ag}$ ← $\mathbf{FramePointer}_{ag}$
<r2> ← (A=$\mathbf{MR}_{tag}$.A, L=$\mathbf{MR}_{tag}$.L, B=$\mathbf{MR}_{tag}$.B, F=0)

17.3.10 `totag <r1> <r2>` – First-Class Tag to Tag

Make *<r2>* be an integer zero tagged by the first-class tag in *<r1>*.

Operands

<r1> – Tag to apply to 0.
<r2> – Resulting 0 tagged with *<r1>*

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = Tag_{ag}

Processor State

PC ← **PC** + **1**
$\textit{<r2>}_{tag}$ ← *<r1>* // Tag comes from payload
$\textit{<r2>}_{ag}$ ← Integer_{ag}
<r2> ← 0

17.3.11 `int <r1> <r2>` – Inspect Tag

Convert first-class tag in *<r1>* to frame-pointer in *<r2>*.
Privilege: Limited to TagSet

Operands

<r1> – First-class Tag to inspect
<r2> – Frame pointer associated with tag

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = Tag_{ag}

Processor State

PC ← **PC** + **1**
$\textit{<r2>}_{tag}$ ← **TMU**
<r2> ← *<r1>*
$\textit{<r2>}_{ag}$ ← FramePtr_{ag}

17.4 Group Management

17.4.1 `regrp <r1> <r2> <r3>` – Regroup Data

Apply group in *<r1>* to the atom in *<r3>* and put combined result into *<r3>*; check that initial group on *<r3>* is same as group on *<r2>*.

[Note: suggested revision 1/14/12 – see discussion with rule idiom in Rule Architecture document –AMD]

Privilege: Only used in specific places in ConcreteWare. See idiom for limiting privilege in Rule Architecture document.

Operands

<r1> – Group to apply, data ignored
<r2> – Group to expect in *<r3>* before change, data ignored
<r3> – Atom that will have its atomic group change to the one in *<r1>*

Processor State

PC $\leftarrow$ **PC** + **1**
<r3>$_{tag}$ $\leftarrow$ **TMU**
<r3>$_{ag}$ $\leftarrow$ *<r1>*$_{ag}$

Error Conditions

<r2>$_{ag} \neq$ *<r3>*$_{ag}$

17.4.2 `ingrp <r1> <r2>` – Inspect Group

Convert group on *<r1>* to integer in *<r2>*
Privilege: unprivileged

Operands

<r1> – Group to inspect, data ignored
<r2> – Result data is group value

Processor State

PC $\leftarrow$ **PC** + **1**
<r2>$_{tag}$ $\leftarrow$ **TMU**
<r2> $\leftarrow$ *<r1>*$_{ag}$

Rules would typically tag *<r2>* with the tag of *<r1>*.

18 Stream OP Codes

18.1 Blocking (Yielding)

18.1.1 `stwy <r1> <r2>` – Stream Write, Yielding

Write atom in *<r1>* to stream id in *<r2>*, yield if stream is full.

Operands

<r1> – Atom to write to stream
<r2> – Output Stream

Allowed Operand Groups

<r1>$_{ag}$ = all but LinearFramePointers, *<r2>*$_{ag}$ = Output Stream Pointer$_{ag}$

Processor State

PC$_{tag}$ ← **TMU**
PC ← **PC + 1**
MR$_{tag}$ ← **mem**[*<r2>*]$_{tag}$ // *for stream in memory case*
MR$_{ag}$ ← **mem**[*<r2>*]$_{ag}$ // need to check empty
MR ← **mem**[*<r2>*]
if (**MR**$_{ag}$ = Empty$_{ag}$)

- **stream**[*<r2>*]$_{tag}$ ← *<r1>*$_{tag}$
- **stream**[*<r2>*] ← *<r1>*
- **stream**[*<r2>*]$_{ag}$ ← *<r1>*$_{ag}$
- *<r2>* ← if (*<r2>*≠ *<r2>*.bound) *<r2>* + 1 else *<r2>*.base
- *<r2>*$_{tag}$ ← **TMU**.2

else

- **TS**.ErrorCode=Stream Blocking

18.1.2 `stwyfree <r1> <r2> [offset]` – Stream Write, Yielding with Free Handling

Write atom in *<r1>* to stream id in *<r2>*, yield if stream is full, branch to *[offset]* if stream has been freed.

Operands

<r1> – Atom to write to stream
<r2> – Output Stream
[offset] – Branch location (within codeblock) if read an End-of-Stream token

Allowed Operand Groups

<r1>$_{ag}$ = all but LinearFramePointers, *<r2>*$_{ag}$ = Output Stream Pointer$_{ag}$

Processor State

PC$_{tag}$ ← **TMU**
PC ← **PC + 1**
MR$_{tag}$ ← **mem**[*<r2>*]$_{tag}$ // *for stream in memory case*
MR$_{ag}$ ← **mem**[*<r2>*]$_{ag}$ // need to check empty
MR ← **mem**[*<r2>*]
if (**MR**$_{ag}$ = Empty$_{ag}$)

- **stream**[*<r2>*]$_{tag}$ ← *<r1>*$_{tag}$
- **stream**[*<r2>*] ← *<r1>*
- **stream**[*<r2>*]$_{ag}$ ← *<r1>*$_{ag}$
- *<r2>* ← if (*<r2>*≠ *<r2>*.bound) *<r2>* + 1 else *<r2>*.base
- *<r2>*$_{tag}$ ← **TMU**.2

else if ($\mathbf{MR}_{ag}$ = FREE_{ag})

- $\mathbf{PC}_{tag} \leftarrow$ **TMU**
- $\mathbf{PC} \leftarrow \mathbf{PC}_{base}$ + *[offset]*

else

- **TS**.ErrorCode=Stream Blocking

18.1.3 `stwly <r1> <r2>` – Stream Write, Linear, Yielding

Move atom in *<r1>* to stream id in *<r2>*, yield if stream is full.

Operands

<r1> – Atom to write to stream
<r2> – Output Stream

Allowed Operand Groups

<r2>_{ag} = Output Stream Pointer_{ag}

Processor State

$\mathbf{PC}_{tag} \leftarrow$ **TMU**
$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$
$\mathbf{MR}_{tag} \leftarrow \mathbf{mem}[\text{<r2>}]_{tag}$ // *for stream in memory case*
$\mathbf{MR}_{ag} \leftarrow \mathbf{mem}[\text{<r2>}]_{ag}$ // need to check empty
$\mathbf{MR} \leftarrow \mathbf{mem}[\text{<r2>}]$
if ($\mathbf{MR}_{ag}$ = Empty_{ag})

- $\mathbf{stream}[\text{<r2>}]_{tag} \leftarrow \text{<r1>}_{tag}$
- $\mathbf{stream}[\text{<r2>}] \leftarrow$ *<r1>*
- $\mathbf{stream}[\text{<r2>}]_{ag} \leftarrow \text{<r1>}_{ag}$
- *<r2>* $\leftarrow$ if (*<r2>* $\neq$ *<r2>*.bound) *<r2>* + 1 else *<r2>*.base
- $\text{<r2>}_{tag} \leftarrow$ **TMU.2**
- $\text{<r1>}_{tag} \leftarrow$ **TMU.3**
- *<r1>* $\leftarrow$ **0**

else

- **TS**.ErrorCode=Stream Blocking

Error Conditions

TID $\neq$ 0 at start of instruction

18.1.4 `stwlyfree <r1> <r2> [offset]` – Stream Write, Linear, Yielding with Free Handling

Move atom in *<r1>* to stream id in *<r2>*, yield if stream is full, branch to *[offset]* if stream has been freed.

Operands

<r1> – Atom to write to stream
<r2> – Output Stream
[offset] – Branch location (within codeblock) if read an End-of-Stream token

Allowed Operand Groups

<r2>$_{ag}$ = Output Stream Pointer$_{ag}$

Processor State

PC$_{tag}$ ← **TMU**
PC ← **PC + 1**
MR$_{tag}$ ← **mem**[*<r2>*]$_{tag}$ // *for stream in memory case*
MR$_{ag}$ ← **mem**[*<r2>*]$_{ag}$ // need to check empty
MR ← **mem**[*<r2>*]
if (**MR**$_{ag}$ = Empty$_{ag}$)

- **stream**[*<r2>*]$_{tag}$ ← *<r1>*$_{tag}$
- **stream**[*<r2>*] ← *<r1>*
- **stream**[*<r2>*]$_{ag}$ ← *<r1>*$_{ag}$
- *<r2>* ← if (*<r2>*≠ *<r2>*.bound) *<r2>* + 1 else *<r2>*.base
- *<r2>*$_{tag}$ ← **TMU.2**
- *<r1>*$_{tag}$ ← **TMU.3**
- *<r1>* ← **0**

else if (**MR**$_{ag}$ = FREE$_{ag}$)

- **PC**$_{tag}$ ← **TMU**
- **PC** ← **PC**$_{base}$ + *[offset]*

else

- **TS**.ErrorCode=Stream Blocking

Error Conditions

TID ≠ 0 at start of instruction

18.1.5 `stry <r1> <r2>` – Stream Read, Yielding

Read from stream id in *<r1>* put atom in *<r2>*, yield if stream is empty.

Operands

<r1> – Input Stream
<r2> – Atom read from stream

Allowed Operand Groups

<r1>$_{ag}$ = Stream Pointer$_{ag}$

Processor State

PC$_{tag}$ ← **TMU**
PC ← **PC + 1**
MR$_{tag}$ ← **stream**[*<r1>*]$_{tag}$
MR$_{ag}$ ← **stream**[*<r1>*]$_{ag}$
MR ← **stream**[*<r1>*]
if (**MR**$_{ag}$ ≠ Empty$_{ag}$)

- *<r2>*$_{tag}$ ← **TMU**
- *<r2>* ← **stream**[*<r1>*]
- *<r2>*$_{ag}$ ← **stream**[*<r1>*]$_{ag}$
- *<r1>* ← if (*<r1>*≠ *<r1>*.bound) *<r1>* + 1 else *<r1>*.base
- *<r1>*$_{tag}$ ← **TMU**.2

else

- **TS**.ErrorCode=Stream Blocking

Error Conditions

TID ≠ 0 at start of instruction

18.1.6 `stryeos <r1> <r2> [offset]` – Stream Read, Yielding with EoS Handling

Read from stream id in *<r1>* put atom in *<r2>*, yield if stream is empty, branch to *[offset]* if End-of-Stream.

Operands

<r1> – Input Stream
<r2> – Atom read from stream
[offset] – Branch location (within codeblock) if read an End-of-Stream token

Allowed Operand Groups

<r1>$_{ag}$ = Stream Pointer$_{ag}$

Processor State

PC$_{tag}$ ← **TMU**
PC ← **PC** + **1**
MR$_{tag}$ ← **stream**[*<r1>*]$_{tag}$
MR$_{ag}$ ← **stream**[*<r1>*]$_{ag}$
MR ← **stream**[*<r1>*]
if ((**MR**$_{ag}$ ≠ Empty$_{ag}$) && (**MR**$_{ag}$ ≠ EOS$_{ag}$))

- *<r2>*$_{tag}$ ← **TMU**
- *<r2>* ← **stream**[*<r1>*]
- *<r2>*$_{ag}$ ← **stream**[*<r1>*]$_{ag}$
- *<r1>* ← if (*<r1>*≠ *<r1>*.bound) *<r1>* + 1 else *<r1>*.base
- *<r1>*$_{tag}$ ← **TMU**.2

elseif (**MR**$_{ag}$ = EOS$_{ag}$)

- **PC**$_{tag}$ ← **TMU**
- **PC** ← **PC**$_{base}$ + *[offset]*

else

- **TS**.ErrorCode=Stream Blocking

Error Conditions

TID ≠ 0 at start of instruction

18.2 Non-Blocking (Branching)

18.2.1 `stwb <r1> <r2> [offset]` – Stream Write, Branching

Write atom in *<r1>* to stream id in *<r2>*, branch to *[offset]* if stream is full.

Operands

<r1> – Atom to write to stream
<r2> – Output Stream pointer
[offset] – Branch destination as unsigned offset from current

Allowed Operand Groups

$\textit{<r1>}_{ag}$ = all but LinearFramePointers, $\textit{<r2>}_{ag}$ = Output Stream Pointer$_{ag}$

Processor State

$\mathbf{PC}_{tag} \leftarrow \mathbf{TMU}$
$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$
$\mathbf{MR}_{tag} \leftarrow \mathbf{mem}[\textit{<r2>}]_{tag}$ // *for stream in memory case*
$\mathbf{MR}_{ag} \leftarrow \mathbf{mem}[\textit{<r2>}]_{ag}$ // need to check empty
$\mathbf{MR} \leftarrow \mathbf{mem}[\textit{<r2>}]$
if ($\mathbf{MR}_{ag}$ = Empty$_{ag}$)

- $\mathbf{stream}[\textit{<r2>}]_{tag} \leftarrow \textit{<r1>}_{tag}$
- $\mathbf{stream}[\textit{<r2>}] \leftarrow \textit{<r1>}$
- $\mathbf{stream}[\textit{<r2>}]_{ag} \leftarrow \textit{<r1>}_{ag}$
- *<r2>* $\leftarrow$ if (*<r2>* $\neq$ *<r2>*.bound) *<r2>* + 1 else *<r2>*.base
- $\textit{<r2>}_{tag} \leftarrow \mathbf{TMU.2}$

else

- $\mathbf{PC}_{tag} \leftarrow \mathbf{TMU}$
- $\mathbf{PC} \leftarrow \mathbf{PC}_{base} + \textit{[offset]}$

Error Conditions

TID $\neq 0$ at start of instruction

18.2.2 `stwlb <r1> <r2> [offset]` – Stream Write, Linear, Branching

Move atom in *<r1>* to stream id in *<r2>*, branch to *[offset]* if stream is full.

Operands

<r1> – Atom to write to stream
<r2> – Output Stream pointer
[offset] – Branch destination as unsigned offset from current

Allowed Operand Groups

$\textit{<r2>}_{ag}$ = Output Stream Pointer$_{ag}$

Processor State

$\mathbf{PC}_{tag} \leftarrow \mathbf{TMU}$
$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$
$\mathbf{MR}_{tag} \leftarrow \mathbf{mem}[\langle r2 \rangle]_{tag}$ // *for stream in memory case*
$\mathbf{MR}_{ag} \leftarrow \mathbf{mem}[\langle r2 \rangle]_{ag}$ // need to check empty
$\mathbf{MR} \leftarrow \mathbf{mem}[\langle r2 \rangle]$
if ($\mathbf{MR}_{ag} = \text{Empty}_{ag}$)

- $\mathbf{stream}[\langle r2 \rangle]_{tag} \leftarrow \langle r1 \rangle_{tag}$
- $\mathbf{stream}[\langle r2 \rangle] \leftarrow \langle r1 \rangle$
- $\mathbf{stream}[\langle r2 \rangle]_{ag} \leftarrow \langle r1 \rangle_{ag}$
- $\langle r2 \rangle \leftarrow$ if ($\langle r2 \rangle \neq \langle r2 \rangle$.bound) $\langle r2 \rangle + 1$ else $\langle r2 \rangle$.base
- $\langle r2 \rangle_{tag} \leftarrow \mathbf{TMU.2}$
- $\langle r1 \rangle_{tag} \leftarrow \mathbf{TMU.3}$
- $\langle r1 \rangle \leftarrow \mathbf{0}$

else

- $\mathbf{PC}_{tag} \leftarrow \mathbf{TMU}$
- $\mathbf{PC} \leftarrow \mathbf{PC}_{base} + [offset]$

Error Conditions

TID $\neq 0$ at start of instruction

18.2.3 `strb <r1> <r2> [offset]` – Stream Read, Branching

Read from stream id in *<r1>* put atom in *<r2>*, branch to *[offset]* if stream is empty.

Operands

<r1> – Input Stream pointer
<r2> – Atom read from stream
[offset] – Branch destination as unsigned offset from current

Allowed Operand Groups

$\langle r1 \rangle_{ag} = \text{Input Stream Pointer}_{ag}$

Processor State

$\mathbf{PC}_{tag} \leftarrow \mathbf{TMU}$
$\mathbf{PC} \leftarrow \mathbf{PC} + \mathbf{1}$
$\mathbf{MR}_{tag} \leftarrow \mathbf{stream}[\langle r1 \rangle]_{tag}$
$\mathbf{MR}_{ag} \leftarrow \mathbf{stream}[\langle r1 \rangle]_{ag}$
$\mathbf{MR} \leftarrow \mathbf{stream}[\langle r1 \rangle]$
if ($\mathbf{MR}_{ag} \neq \text{Empty}_{ag}$)

- $\langle r2 \rangle_{tag} \leftarrow \mathbf{TMU}$
- $\langle r2 \rangle \leftarrow \mathbf{stream}[\langle r1 \rangle]$
- $\langle r2 \rangle_{ag} \leftarrow \mathbf{stream}[\langle r1 \rangle]_{ag}$
- $\langle r1 \rangle \leftarrow$ if ($\langle r1 \rangle \neq \langle r1 \rangle$.bound) $\langle r1 \rangle + 1$ else $\langle r1 \rangle$.base
- $\langle r1 \rangle_{tag} \leftarrow \mathbf{TMU}.2$

else

- **PC** $\leftarrow$ **PC**$_{base}$ + *[offset]*

Error Conditions

TID $\neq 0$ at start of instruction

19 Transaction OP Codes

19.0.4 `transbegin [id]` – Begin Transaction

Begin a transaction. Operations between the begin and the end should appear to occur atomically.
Privilege: unprivileged – any authority can use

Operands

[id] – ID for transaction begun to match transbegin/transend pairs

Allowed Operand Groups

N/A

Processor State

TID $\leftarrow$ *[id]*
PC$_{tag}$ $\leftarrow$ **TMU**
PC $\leftarrow$ **PC** + **1**

Error Conditions

TID $\neq 0$ at start of instruction
transaction attempts to execute more than TXN_MAXINSTR instrucitons
transaction attempts to perform more than TXN_MAXWRITE writes
transaction attempts to perform gates calls (or other unallowed operations)
transaction attempts to perform a backward control transfer

19.0.5 `transend [id]` – End Transaction

End a transaction. Commit state changes from transaction.
Privilege: unprivileged – any authority can use

Operands

[id] – ID of transaction to end to match transbegin/transend pairs

Allowed Operand Groups

N/A

Processor State

PC$_{tag}$ ← **TMU**
PC ← **PC + 1**
// potentially nothing
// in practice, release write buffer to flush changes to memory and write back Thread Frame changes
TID ← 0

Error Conditions

TID ≠ *[id]* at start of instruction

20 Miscellaneous OP Codes

20.0.6 `nop` – No Operation

Do nothing.

Operands

None

Processor State

PC ← **PC + 1**

20.0.7 `halt` – Halt Processor

Causes the processor to halt.

Operands

None

Processor State

21 Field Sizes and Encodings

21.1 Definition for Tools

Canonical definitions used as single source for assembler, safe-sim, and bluespec implementation are in `isa/tools/safe-isa/Setup.hs` and `isa/tools/safe-isa/OneSource.hs`.

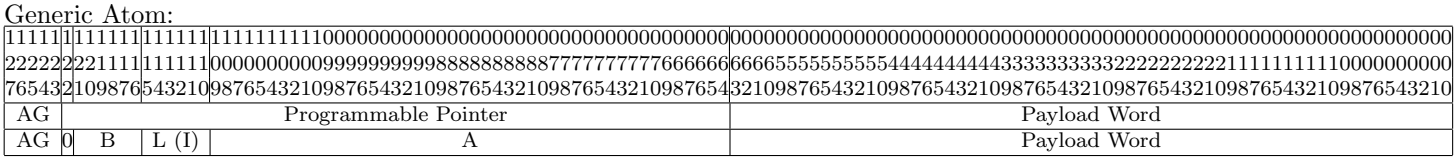
To get a dump of symbols defined for the assembler:

```
safe-asm --print-hardware-symbols
```

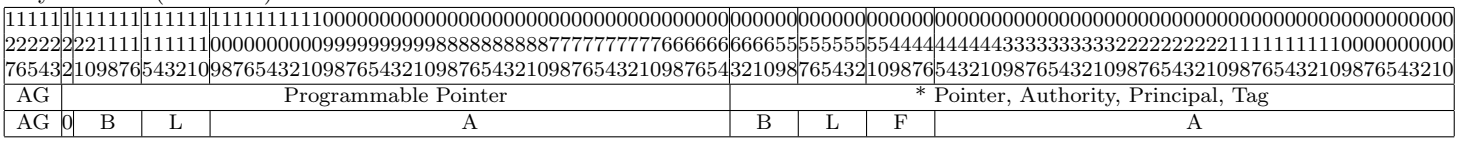
21.2 Definitions

Field	Bits	Explanation
Atom	128	Metadata + Payload Word
Payload Word	64	holds one double, int64, Fat Pointer
Metadata	64	AG + Programmable Pointer
AG	5	holds up to 32 (currently 17 defined)
Fat Pointer	64	(B,L,F,A) or (B,M,I,A)
Fat Pointer: A	46	
Fat Pointer: F (I)	6	
Fat Pointer: B	6	
Fat Pointer: L (M)	6	
Programmable Pointer	59	deliberately fudged so AG+Programmable Pointer is 64b
A in Programmable Pointer	46	
F (M) in Programmable Pointer	0	assume always at base, gives us back 5b for AG plus one extra
Instruction	32	Allow more compact packing in some encodings
Opcode	7	support up to 128 (might bump to 8)
Register	5	support up to 32
Immediate	10–25	depends on opcode

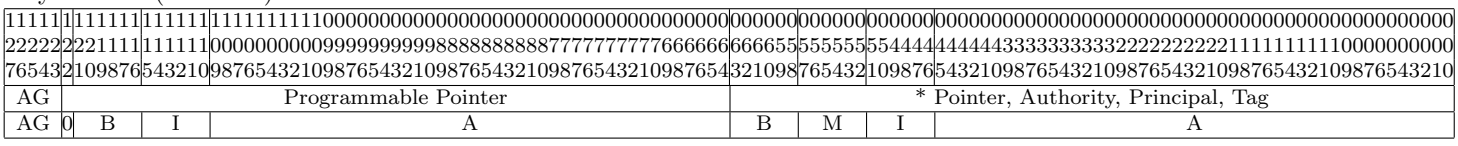
Generic Atom:



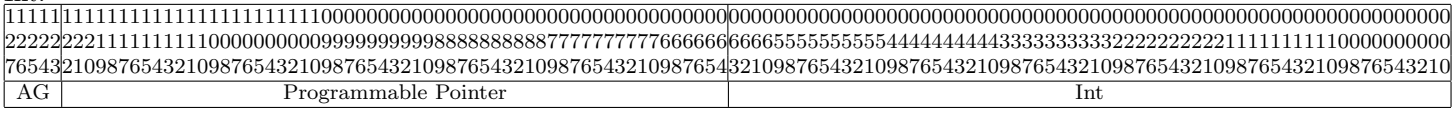
Any Pointer (B L F A):



Any Pointer (B M I A):



Int:



Double Float:

11111	11111111111111111111111100000000000000000000000000000000000000	00000000000000000000000000000000000000000000000000000000000000
22222	22211111111110000000009999999998888888887777777776666666	66665555555554444444443333333332222222221111111110000000000
76543	21098765432109876543210987654321098765432109876543210987654	32109876543210987654321098765432109876543210987654321098765432
AG	Programmable Pointer	Double Float

Generic Instruction:

11111	11111111111111111111111100000000000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000
22222	22211111111110000000009999999998888888887777777776666666	66665555555554444444443333333332	22222222222211111111110000000000
76543	21098765432109876543210987654321098765432109876543210987654	32109876543210987654321098765432	10987654321098765432109876543210
AG	Programmable Pointer	Unused	Instruction

Notes:

- Encoding for opcode should be chosen so can represent instruction groups that will have same TMU rules by subset of bits. *E.g.*, all the 3-address arithmetic ops should have the same distinguishable prefix. See Section 22.
- In future, may want to consider 128b payload; demands packing more things into word (*e.g.* two doubles).
- When reify Programmable Pointer as Frame F is inserted as 0.
- Could reduce Programmable Pointer bits by:
  - Enforce all tag objects/frames are of same size and point to entire object. Consequently, remove L, B, F from Programmable pointer – add in when reify Programmable Pointer as Frame Pointer.
  - Limit Programmable Pointer address space to even fewer bits
- With Programmable Pointer A limited to 27b, could reduce Metadata to 32b.
- Since Atoms are our smallest addressable unit, we cannot pack multiple instructions into an atom without introducing PC complexities (need to keep track of PC being in the middle of an atom).
- A code block likely has:
  - All instructions tagged in same way (same AG, Programmable Pointer)
  - All instruction preceding any atoms in the code block
- Should be able to have a compressed code block version where pack instructions into 32b sequences that share a tag. Useful for out-of-memory case. Adds complexity and need to verify that expansion does not introduce vulnerabilities. Will need something like this for code signature and intern in any case.

Instruction Sub Fields:

Instr Class	3322222109876543210	22222	11111 98765	11111 43210	0000000000 9876543210
N	opcode	0			
S	opcode	0	src1	0	
SS	opcode	0	src1	src2	0
SD	opcode	dst	src1	0	
SSD	opcode	dst	src1	src2	0
SSDO	opcode	dst	src1	src2	imm
O	opcode	imm			
SO	opcode	0	src1	imm	
DO	opcode	dst	imm		
SSO	opcode	0	src1	src2	imm
SDO	opcode	dst	src1	imm	

Notes:

- Should we try to pick a single location for dst to avoid the need to multiplex the destination into the register file write address based on instruction? This potentially also shows up in pipeline bypassing once we pipeline the design.
- Should we try to pick a single location for the pointer used to reference memory? This is showing up as complexity (and multiplexing) in the logic. Similarly, it means the register that gets the forwarding pointer is also dependent on the instruction.

Current plan is to make changes like this in the near future.

22 Instruction Groups

To simplify the TMU (and our reasoning), we can identify groups of instructions that have similar use and interpertation of their operands.

Group Name	Instructions
arith2s1d	add sub and or shl shr test mul
none	nop
jmp	jmp
yield	yield
grtn	grtn
ptr2s1d	offp
arith1s1d	invert
cprr	cprr
ptr1s1d	offlp
gatecalls	gcall gjmp
branch cond	beq bneq bneg bpos
stwy	stwy
stwb	stwb
stry	stry
strb	strb
grtn	grtn
mvrr	mvrr
mvmr	mvmr
cpmr	cpmr
mvrm	mvrm
cprm	cprm
call	call
lcfp	lcfp
give-time	give-time
recover-time	recover-time
read-time	read-time
runt	runt
gate	gate
fjmp	fjmp
retag	retag
seta	seta
intag	intag
halt	halt
framptr	framptr
makep	makep
tmul	tmul
tmuu	tmuu
tmurc	tmurc
newt	newt
ingrp	ingrp
regrp	regrp
transactions	transbegin transend
offtp	offtp
cphr	cphr
cpar	cpar
cpio	cpio
cpmt	cpmt

Group Name	Instructions
inp	inp
resumet	resumet
totag	totag
rflct	rflct
gfcall	gfcall
fphash	fphash
rflctp	rflctp
retagpc	retagpc
settag	settag
maket	maket
tagof	tagof

23 Implementation Status

23.1 Instructions

This chart describes which instructions have been implemented in the various safe system tools.

Key

Y	Yes, implemented
M	implemented but not tested
N	Not implemented

OpCode	safe-asm/safe-isa	Coq	SafeSim	Bluespec	Breeze Compiler Uses	ConcreteWare Uses	TestCase
acall *<r1> <r2>*	Y	N	N	N		N	none
add *<r1> <r2> <r3>*	Y	Y	Y	Y	Y	Y	initial.asm
addf *<r1> <r2> <r3>*	N	N	N	N		N	?
and *<r1> <r2> <r3>*	Y	Y	Y	Y		?	arith.asm
basep *<r1> <r2>*	N	N	N	N		N	?
bcall *<r1> <r2> <r3> [result-reg]*	N	N	N	N		N	?
beq *<r1> [offset]*	Y	Y	Y	Y	Y	?	summation.asm
bgacall *<r1> <r2> <r3> [result-reg]*	N	N	N	Y		N	?
bgcall *<r1> <r2> <r3> [result-reg]*	N	N	N	Y		N	?
bgfcall *<r1> <r2> <r3> [wmask]*	N	N	N	N		N	?
bne *<r1> [offset]*	Y	Y	Y	Y		?	?
bneg *<r1> [offset]*	Y	Y	Y	Y		?	?
brtn *<r1>*	N	N	N	Y		N	?
call *<r1>*	Y	N	Y	Y		N	call.asm
clear *<r1>*	N	N	N	N		N	?
clearregs *<r1>*	N	N	N	N		N	?
cpar *<r1> <r2>*	Y	N	Y	Y		Y	
cphr *<r1> <r2>*	Y	N	Y	Y		Y	
cpio *<r1> <r2>*	Y	N	Y	Y		Y	
cpmr *<r1> <r2>*	Y	Y	Y	Y	Y	Y	cpxx.asm
cpmt *<r1> <r2>*	Y	N	Y	Y		Y	
cprm *<r1> <r2>*	Y	Y	Y	Y	Y	Y	cpxx.asm
cprr *<r1> <r2>*	Y	Y	Y	Y	Y	Y	cpxx.asm
endt	Y	N	N	Y	Y	?	?
fjmp *<r1>*	Y	N	Y	Y		N	jump.asm
fphash *<r1> <r2>*	Y	N	N	N		Y	none
framptr *<r1> <r2> <r3>*	Y	N	Y	Y		Y	framptr.asm
gacall *<r1> <r2>*	Y	N	N	Y		N	summation_gacall.asm
gate *<r1> <r2> <r3>*	Y	Y	Y	Y	Y	Y	gate.asm
gatele *<r1> <r2> <r3>*	Y	N	N	N		N	none
gcall *<r1>*	Y	Y	Y	Y	Y	Y	gate.asm
gfcall *<r1> <r2> <r3>*	N	N	N	N		N	?
gfwrite *<r1> <r2> <r3>*	Y	N	N	N		N	none
give-time *<r1>*	Y	N	N	Y		Y	?
gjmp *<r1>*	Y	N	Y	Y	Y	N	?
grtn	Y	N	Y	Y		Y	gate.asm
halt	Y	Y	Y	Y	Y	?	?

OpCode	safe-asm/safe-isa	Coq	SafeSim	Bluespec	Breeze Compiler Uses	ConcreteWare Uses	TestCase
ina *<r1> <r2>*	Y	N	N	Y		N	none
ingrp *<r1> <r2>*	Y	N	Y	Y		N	?
inp *<r1> <r2>*	Y	N	N	Y		N	none
int *<r1> <r2>*	Y	N	Y	N		N	none
intag *<r1> <r2>*	Y	N	N	Y		N	tags.asm
jmp *[offset]*	Y	Y	Y	Y		N	?
lcall *<r1> <r2>*	N	N	N	N		N	none
lcfp *<r1> [offset]*	Y	Y	Y	Y	Y	Y	cpxx.asm
livemask *<r1>*	N	N	N	N		N	?
lowera *<r1>*	Y	N	N	N		N	none
mul *<r1> <r2> <r3>*	Y	N	N	Y	Y	N	arith.asm
mvmr *<r1> <r2>*	Y	Y	Y	Y	Y	Y	mvxx.asm
mvrm *<r1> <r2>*	Y	Y	Y	Y	Y	Y	mvxx.asm
mvrr *<r1> <r2>*	Y	Y	Y	Y	Y	Y	mvxx.asm
newt *<r1> <r2>*	Y	N	Y	Y		N	?
nop	Y	Y	Y	Y		N	?
not *<r1> <r3>*	N	N	N	N		N	?
offlp *<r1> <r2>*	Y	N	Y	Y		N	?
offp *<r1> <r2> <r3>*	Y	N	Y	Y	Y	Y	summation.asm
offtp *<r1> <r2>*	Y	N	Y	Y		Y	?
or *<r1> <r2> <r3>*	Y	Y	Y	Y		?	arith.asm
raisea *<r1>*	Y	N	N	N		N	none
rcall *<r1> <r2>*	N	N	N	N		N	none
read-time *<r1>*	Y	N	N	Y		N	?
recover-time *<r1>*	Y	N	N	Y		N	?
regrp *<r1> <r2>*	Y	N	Y	Y		Y	?
resumet *<r1> [entry]*	Y	N	N	Y		Y	?
retag *<r1> <r2>*	Y	N	Y	Y		Y	tags.asm
retagpc *<r1>*	Y	N	N	Y		N	none
rflct *<r1> <r2>*	Y	N	N	Y		Y	none
runt *<r1>*	Y	N	Y	Y		Y	?

OpCode	safe-asm/safe-isa	Coq	SafeSim	Bluespec	Breeze Compiler Uses	ConcreteWare Uses	TestCase
seta *<r1>*	Y	N	N	Y		N	?
settag *<r1> <r2>*	Y	N	Y	Y		?	none
shl *<r1> <r2> <r3>*	Y	Y	Y	Y		?	arith.asm
shr *<r1> <r2> <r3>*	Y	Y	Y	Y		?	arith.asm
sizep *<r1> <r2>*	N	N	N	N		N	?
strb *<r1> <r2> [offset]*	Y	Y	Y	Y		?	stream.asm
stry *<r1> <r2>*	Y	Y	Y	Y		?	Y
stryeos *<r1> <r2> [offset]*	N	N	N	N		?	N
stwb *<r1> <r2> [offset]*	Y	Y	Y	Y		?	stream.asm
stwlb *<r1> <r2> [offset]*	Y	N	N	N		N	none
stwly *<r1> <r2>*	Y	N	N	N		N	?
stwlyfree *<r1> <r2> [offset]*	N	N	N	N		?	?
stwy *<r1> <r2>*	Y	Y	Y	Y		Y	?
stwyfree *<r1> <r2> [offset]*	N	N	N	N		?	?
sub *<r1> <r2> <r3>*	Y	Y	Y	Y	Y	?	arith.asm
tagof *<r1> <r2>*	Y	N	N	Y		?	none
tagofpc *<r1>*	N	N	N	N		?	none
tcall *<r1> <r2>*	N	N	N	N		N	none
test *<r1> <r2> <r3>*	Y	N	Y	Y	Y	?	?
testgrp *<r1> <r2> [grpid]*	Y	N	N	Y		?	testgrp.asm
tmul *<r1> <r2> <r3>*	Y	N	N	Y		Y	?
tmurc *<r1> <r2>*	Y	N	N	Y		Y	?
tmuu *<r1>*	Y	N	N	Y		Y	?
totag *<r1> <r2>*	Y	N	N	Y		?	none
transbegin *[id]*	Y	N	N	Y		Y	trans_test.asm
transend *[id]*	Y	N	N	Y		Y	trans_test.asm
trequire *<r1>*	N	N	N	N		N	none
trtn	N	N	N	N		N	none
writemask *<r1>*	N	N	N	N		N	?
xor *<r1> <r2> <r3>*	N	N	N	N		N	?
yield	Y	N	Y	Y		Y	?
yield2 *<r1> [entry]*	N	N	N	N		N	?

23.2 Atomic Groups

This chart describes which atomic groups have been implemented in the various safe system tools.

Atomic Group	safe-asm/safe-isa	Coq	SafeSim	Bluespec	TestCase
Frame Pointer	Y	Y	Y	Y	?
Constant Frame Pointer	N	N	N	N	?
Linear Frame Pointer	Y	N	Y	Y	?
Instruction Pointer	Y	Y	Y	Y	?
Gate Pointer	Y	Y	Y	Y	?
Stream Read Pointer	Y	N	Y	Y	?
Stream Write Pointer	Y	N	Y	Y	?
Thread Pointer	Y	Y	Y	Y	?
Forwarding Pointer	Y	N	Y	N	?
Double (Double Float)	Y	N	Y	Y	?
Integer	Y	Y	Y	Y	?
Instruction	Y	Y	Y	Y	?
Authority	Y	Y	Y	Y	?
Principal	Y	N	Y	Y	?
Uninitialized	Y	N	Y	Y	?
Error	Y	N	Y	Y	?
Tag	Y	N	Y	Y	?
Empty	Y	Y	Y	Y	?
EOS	N	N	N	N	?
FREE	N	N	N	N	?