UpDown Instruction Set Architecture
Description v0.95 (9/29/21)

Table of Contents:
Preface 3
UpStream Lane Overview 3
  Upstream Accelerator Lane 3
  Terms and Definitions 4
UpStream Code Memory Map 5
UpStream Thread State (software thread) 7
Updates to UDP ISA 8
  Transition Primitives 8
  Actions 10
  New UpStream Actions 14
    Conditional Branch instructions 14
    Instructions for Fast Messaging 14
    Event Update instructions: 20
    Synchronization instructions 21
    Additional Move Instructions 21
    Thread Switch and Terminate Instructions 21
UpStream Thread Lifecycle 22
UpStream Lane MicroArchitecture 24
System Memory Map 25
UpStream Examples 27
  Example1: Strided Address Generation 27
  Example2: Tail Recursion using threads 28
  Example3: Tail Recursion using threads on successive lanes 29
  Example4: Fast Messaging - Stream DRAM data into ScratchPad 30
  Example5: Fast Messaging - Stream Scratch Pad data into DRAM 31
  Example6: Binary Recursion 32
  Example7: fork/join 33
  Example8: Triangle Counting 36
References 38
Preface

This document contains the UpStream Processor hardware architecture and the SW interface to it. It is described as an update to the UDP architecture and ISA found in [1]. The architectural updates and ISA additions that support these updates are described in detail and examples are added to demonstrate performance and use cases.

UpStream Lane Overview

The UpStream Lane is based on the UDP Lane architecture. UpStream adds event dispatch, efficient synchronization and fast messaging support to the existing UDP lane. Section 5 of the UDP Instruction Set Architecture document describes the UDP lane architecture in detail. In the section below, we describe the additions/modifications in the UpStream Lane over the UDP architecture.

Upstream Accelerator Lane

![UpStream Lane Architecture](image)

1. Upstream features – event dispatch, synchronization, fast messaging

Fig 1: UpStream Lane Architecture
Terms and Definitions

The following are the terms and definitions associated with new features in the UpStream design.

**Event**: An event consists of an event word and associated operands.

**Event Word[32]**: An event word is a 32 bit value consisting of
- Event program (event_base [8?]), event_label[8], thread_id [8], lane_id[8])
  - event_base, event_label determine the address of the Upstream transition to be executed (can be thought of as function entry point)
  - Base could index a lookup table or <base,event> could be concatenated for a transition address
  - thread_id - specifies the thread that the current event belongs to
  - lane_id - lane ID of originating lane - follows same naming as dest ID under “Fast Messaging”

Fig 2: Event Word

**Operands**: Event Operands are arguments that are used by the codelet triggered into execution by the event word. The operands are stored in the Operand buffer described below.

**EventQueue**: (EvQ) The Event Queue contains the sequence of event words waiting to be processed by the UpStream Lane. EvQ.TOP points to the head of the event queue and is also the current event being processed by the UpStream Lane.

**OperandBuffer**: (OB - size opsz) The Operand Buffer is an addressable FIFO buffer that holds the operands for the current event. Each event could be associated with zero to many operands. The operand buffer is addressed relative to the FIFO head. So, OB_0 is the head of the FIFO, OB_1 is head+1 and so on. On yield(described later) it is SW’s responsibility to pop the operands corresponding to the current event to ensure correctness [currently]
ExtendedAddressRegisters: (EAR[0-3]) are 64-bit address registers that enable generation of DRAM addresses with the EAR registers acting as base addresses. Offsets from this base address can come from any of the UDPR/OB registers or an imm value.

AddressGenerationUnit: (AGU) The Address Generation Unit generates DRAM addresses using the EAR[0-3] registers as base and an offset address whose value is specified by the “Ra” field in the send instruction described later.

DRAM Address = EAR[0|1|2|3] + offset
UpStream Code Memory Map

The UpStream kernel is laid out in memory in codelets. Each instruction in the codelet is placed contiguously although the codelets themselves need not be contiguous in memory. The transitions for all codelets are placed contiguously with each transition pointing to its corresponding block of actions. Action blocks can be shared between transitions and can be placed separately to avoid replication of same actions in multiple codelets.

![Sample Code Memory Map for an Upstream kernel](image)

Fig 4: Sample Code Memory Map for an Upstream kernel
UpStream Thread State (software thread)

The Upstream software thread state is defined by the following set of values/registers (ControlRegister, GPR’s from the UDP, and EventQ.top, OperandQ.topS and ThreadState).

Fig 5: UpStream SW Thread State

1. **ControlRegister (CR)** - The Control Register from UDP contains the following words.
   - UIP[15:0] - UDP instruction-word pointer register that points to the word address of the current executing UDP instruction (transition primitive or action primitive)
   - lanes_fsm [3:0] - Current internal state of the UDP lane (refer to State property description)
   - SBP[31:0] - 32-bit input stream pointer. In vector-register mode, SBP points to the stream buffer. In local memory mode, SBP points to the local memory byte-address.
   - issue_width[3:0] - determines next issued input word width. input word = [SBP:SBP+issue_width]
   - advance_width[2:0] -
   - rdMode [0] - 0 - vector register mode, 1 - local memory mode
2. Thread State Register (TS) - thread state register is the entry corresponding to the current thread in the thread state table. It contains the following information for the current thread:
   - PTID[31:16] - 8 bit parent thread ID to identify thread that created current thread
   - TID[15:8] - 8 bit thread ID to identify the active thread

3. General Purpose Registers (UDPR_0 - UDPR_n) - n UDP registers allocated to the thread.

4. EventQ.Top (EQT) - This is the current event word being processed by the thread. EQT can be used as the template to create new events. EQT points to the head of the Event Queue.

5. Operand Buffer registers (OB_0 - OB_m) - m operand buffers that correspond to the current event of the thread being processed. This points to the m registers from the head of the Operand Buffer.

In general, actions can use value specifiers that name state from any of the software thread state listed above.

Updates to UDP ISA

Transition Primitives

UDP Supported 7 transition primitives. We add an additional transition primitive to support Event transitions. So in total there are 8 transition primitives as below.

1) labeled_TX targets fast symbol based transition executed in single cycle,
2) majority_TX targets to reduce the number of transitions within a single state,
3) default_TX targets to reduce the number of transitions among states,
4) epsilon_TX targets providing NFA-like concurrent activations,
5) common_TX targets at providing “don’t care” transition for efficient representation of the string distance,
6) flagged_TX targets at enabling control-flow aware state transition,
7) refill_TX targets at variable-sized symbol execution.
8) Event_TX - allows for event based transitions from NULL to valid state
9) Yield_TX - transition on yield/yield_terminate to NULL.
Fig 6: UpStream Transitions

<table>
<thead>
<tr>
<th>TX Type</th>
<th>Transition Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Labeled_TX</td>
<td>[current_active_state-----input_word------&gt;next_active_state]</td>
</tr>
<tr>
<td>Default_TX</td>
<td>[current_active_state-------&gt;default_state]</td>
</tr>
<tr>
<td>Flagged_TX</td>
<td>[current_active_state-----UDPR0------&gt;next_active_state]</td>
</tr>
<tr>
<td>Epsilon_TX</td>
<td>[next_active_state_1------&gt;next_active_state_2]</td>
</tr>
<tr>
<td>Majority_TX</td>
<td>[current_active_state---- NOT IN [input_word_1, input_word_2, ..., input_word_X] ----&gt;next_active_state]</td>
</tr>
<tr>
<td>Common_TX</td>
<td>[current_active_state--------&gt;next_active_state]</td>
</tr>
<tr>
<td>Refill_TX</td>
<td>[current_active_state-----input_word------&gt;next_active_state] SBP rollback x bits. x = 1-8, which is specified in the transition</td>
</tr>
<tr>
<td>Event_TX</td>
<td>[null_state ---EvQ.top------&gt;next_active_state]</td>
</tr>
<tr>
<td>yield_TX</td>
<td>[current_active_state --- yield/yield_terminate ----&gt;null_state]</td>
</tr>
</tbody>
</table>
**Actions**

New instructions in UpStream: Green

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_state_property $property</code></td>
<td>set next_active_state’s property ($type, $value)</td>
</tr>
<tr>
<td><code>fork_state $state_ident, $property</code></td>
<td>create a new next_activation = ($state_ident, $property, lane.lmBase)</td>
</tr>
<tr>
<td><code>set_issue_width $width</code></td>
<td>CR.issue_width = $width</td>
</tr>
<tr>
<td><code>put_2bytes_imm UDPR, $bytes</code></td>
<td>LM[UDPR].write2Byte($bytes); UDPR+=2</td>
</tr>
<tr>
<td><code>put_1byte_imm UDPR, $bytes</code></td>
<td>LM[UDPR] = write1Byte($bytes); UDPR+=1</td>
</tr>
<tr>
<td><code>put_bytes UDPRs, UDPRd, $len</code></td>
<td>X = $len; LM[UDPRd] = writeXByte(UDPRs); UDPRd+=X;</td>
</tr>
<tr>
<td><code>get_bytes UDPRs, UDPRd, $len</code></td>
<td>X = $len; UDPRd = LM[UDPRs].readXByte(); UDPRs+=X;</td>
</tr>
<tr>
<td><code>put_bits UDPR, $bits, $len</code></td>
<td>X = $len; LM[UDPR].writeXBit($bits); UDPR+=X</td>
</tr>
<tr>
<td><code>get_bits UDPRs, UDPRd, $len</code></td>
<td>X = $len; UDPRs = LM[UDPRs].readXBit(); UDPRs+=X</td>
</tr>
<tr>
<td><code>compare_string Rs, Rt, UDPRd</code></td>
<td>UDPRd←compare_str(Rs, Rt). No update on Rs, Rt.</td>
</tr>
<tr>
<td><code>copy UDPRs, UDPRt, UDPRd</code></td>
<td>copy(UDPRs, UDPRd, UDPRt). update UDPRs, UDPRt and UDPRd. UDPRd=0 if copy finish</td>
</tr>
<tr>
<td><code>copy_imm UDPRs, UDPRd, $length</code></td>
<td>copy(UDPRs, UDPRd, $length). update UDPRs and UDPRd.</td>
</tr>
<tr>
<td><code>lshift UDPRs, UDPRd, $shift</code></td>
<td>UDPRd =UDPRs &lt;&lt; $shift</td>
</tr>
<tr>
<td><code>rshift UDPRs, UDPRd, $shift</code></td>
<td>UDPRd =UDPRs &gt;&gt; $shift</td>
</tr>
<tr>
<td><code>lshift_or UDPRs, UDPRd, $shift</code></td>
<td>UDPRd =UDPRd</td>
</tr>
<tr>
<td><code>rshift_or UDPRs, UDPRd, $shift</code></td>
<td>UDPRd =UDPRd</td>
</tr>
<tr>
<td><code>lshift_and UDPRs, UDPRd, $shift</code></td>
<td>UDPRd =UDPRd &amp; (UDPRs &lt;&lt; $shift)</td>
</tr>
<tr>
<td><code>rshift_and UDPRs, UDPRd, $shift</code></td>
<td>UDPRd =UDPRd &amp; (UDPRs &gt;&gt; $shift)</td>
</tr>
<tr>
<td><code>lshift_or_imm UDPRs, UDPRd, $shift, $imm</code></td>
<td>UDPRd =$imm</td>
</tr>
<tr>
<td><code>rshift_or_imm UDPRs, UDPRd, $shift, $imm</code></td>
<td>UDPRd =$imm</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>lshift_and_imm UDPRs, UDPRd, $shift, $imm</td>
<td>UDPRd =$imm &amp; (UDPRs &lt;&lt; $shift)</td>
</tr>
<tr>
<td>rshift_and_imm UDPRs, UDPRd, $shift, $imm</td>
<td>UDPRd =$imm &amp; (UDPRs &gt;&gt; $shift)</td>
</tr>
<tr>
<td>lshift_add_imm UDPRs, UDPRd, $shift, $imm</td>
<td>UDPRd =$imm + (UDPRs &lt;&lt; $shift)</td>
</tr>
<tr>
<td>lshift_sub_imm UDPRs, UDPRd, $shift, $imm</td>
<td>UDPRd = (UDPRs &lt;&lt; $shift) - $imm</td>
</tr>
<tr>
<td>rshift_add_imm UDPRs, UDPRd, $shift, $imm</td>
<td>UDPRd =$imm + (UDPRs &gt;&gt; $shift)</td>
</tr>
<tr>
<td>rshift_sub_imm UDPRs, UDPRd, $shift, $imm</td>
<td>UDPRd = (UDPRs &gt;&gt; $shift) - $imm</td>
</tr>
<tr>
<td>rshift UDPRs, UDPRd, $imm</td>
<td>UDPRd = (UDPRs &gt;&gt; $imm)</td>
</tr>
<tr>
<td>lshift UDPRs, UDPRd, $imm</td>
<td>UDPRd = (UDPRs &lt;&lt; $imm)</td>
</tr>
<tr>
<td>arithrshift UDPRs, UDPRd, $imm</td>
<td>UDPRd = (UDPRs &gt;&gt;&gt; $imm)</td>
</tr>
<tr>
<td>rshift_t UDPRs, UDPRt, UDPRd</td>
<td>UDPRd = (UDPRs &gt;&gt; UDPRt)</td>
</tr>
<tr>
<td>lshift_t UDPRs, UDPRt, UDPRd</td>
<td>UDPRd = (UDPRs &lt;&lt; UDPRt)</td>
</tr>
<tr>
<td>arithrshift_t UDPRs, UDPRt, UDPRd</td>
<td>UDPRd = (UDPRs &gt;&gt;&gt; UDPRt)</td>
</tr>
<tr>
<td>hashsb32 UDPRd, $HTBASE</td>
<td>UDPRd = hash(sb[SBP]) + $HTBASE or UDPRd = hash(LM[SBP]) + $HTBASE</td>
</tr>
<tr>
<td>addi Rs, Rd, $imm</td>
<td>Rd ← Rs + $imm</td>
</tr>
<tr>
<td>subi Rs, Rd, $imm</td>
<td>Rd ← Rs - $imm</td>
</tr>
<tr>
<td>add Rs, Rt, Rd</td>
<td>Rd ← Rs + Rt</td>
</tr>
<tr>
<td>sub Rs, Rt, Rd</td>
<td>Rd ← Rs - Rt</td>
</tr>
<tr>
<td>mov_lm2reg UDPRs, UDPRd, $bytes</td>
<td>UDPRd ← LM[DS+UDPRs : DS+UDPRs+$bytes-1] &amp; mask $bytes=1, mask = 0xff $bytes = 2, mask = 0xffff $bytes =3, mask = 0xffffffff $bytes =4, mask = 0xffffffff</td>
</tr>
<tr>
<td>mov_reg2lm UDPRs, UDPRd, $bytes</td>
<td>LM[DS+UDPRd : DS+UDPRd +$bytes-1] ← UDPRs &amp; mask</td>
</tr>
<tr>
<td>mov_sb2reg UDPRd</td>
<td>UDPRd ← stream_buffer[SBP:SBP+CR.issue]</td>
</tr>
<tr>
<td>mov_reg2reg UDPRs, UDPRd</td>
<td>UDPRd ← UDPRs</td>
</tr>
<tr>
<td>mov_imm2reg UDPRd, $imm</td>
<td>UDPRd ← $imm</td>
</tr>
<tr>
<td>comp_lt UDPRs, UDPRd, $imm</td>
<td>UDPRd ← UDPRs &lt; $imm?</td>
</tr>
<tr>
<td><strong>Operation</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
</tr>
<tr>
<td><strong>comp_gt</strong> UDPRs, UDPRd, $imm**</td>
<td>UDPRd ← UDPRs &gt; $imm?</td>
</tr>
<tr>
<td><strong>comp_eq</strong> UDPRs, UDPRd, $imm**</td>
<td>UDPRd ← UDPRs = $imm?</td>
</tr>
<tr>
<td><strong>compreg_lt</strong> UDPRs, UDPRt, UDPRd</td>
<td>UDPRd ← UDPRs &lt; UDPRt?</td>
</tr>
<tr>
<td><strong>compreg_gt</strong> UDPRs, UDPRt, UDPRd</td>
<td>UDPRd ← UDPRs &gt; UDPRt?</td>
</tr>
<tr>
<td><strong>compreg_eq</strong> UDPRs, UDPRt, UDPRd</td>
<td>UDPRd ← UDPRs = UDPRt?</td>
</tr>
<tr>
<td><strong>TranCarry_goto</strong> $BLOCK_ID</td>
<td>UIP ← address of symbolic shared action block $BLOCK_ID. In Machine Code, it is implemented as a transition primitive</td>
</tr>
<tr>
<td><strong>goto</strong> $BLOCK_ID</td>
<td>UIP ← address of symbolic shared action block $BLOCK_ID.</td>
</tr>
<tr>
<td><strong>refill</strong> $imm</td>
<td>When stage finish, SBPB ← SBPB - $imm + CR.Advcance</td>
</tr>
<tr>
<td><strong>bitwise_and_imm</strong> UDPRs, UDPRd, $imm</td>
<td>UDPRd ← UDPRs &amp; $imm</td>
</tr>
<tr>
<td><strong>bitwise_and</strong> UDPRs, UDPRt, UDPRd</td>
<td>UDPRd ← UDPRt &amp; UDPRs</td>
</tr>
<tr>
<td><strong>bitwise_or_imm</strong> UDPRs, UDPRd, $imm</td>
<td>UDPRd ← UDPRs</td>
</tr>
<tr>
<td><strong>bitwise_or</strong> UDPRs, UDPRt, UDPRd</td>
<td>UDPRd ← UDPRs</td>
</tr>
<tr>
<td><strong>bne</strong> R1, R2, target</td>
<td>Branch to target if R1 != R2. R1, R2 are registers in OB</td>
</tr>
<tr>
<td><strong>beq</strong> R1, R2, target</td>
<td>Branch to target if R1 == R2. R1, R2 are registers in OB</td>
</tr>
<tr>
<td><strong>bgt</strong> R1, R2, target</td>
<td>Branch to target if R1 &gt; R2. R1, R2 are registers in OB</td>
</tr>
<tr>
<td><strong>blt</strong> R1, R2, target</td>
<td>Branch to target if R1 &lt; R2. R1, R2 are registers in OB</td>
</tr>
<tr>
<td><strong>send</strong> Re, Rd, Ra, $sz, $rw</td>
<td>Re - event_word that message return should trigger Rd - destination ID Ra - Register for Addressing $sz - Number of bytes to read/write (imm value) $rw - read / write bit (imm value) (all values for Re, Rd, Ra could come from software thread state -- with limits for encoding)</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>cmpswp r7, mem, imm1, imm2</td>
<td><code>mem = imm2 if mem == imm1 (r7 = mem)</code></td>
</tr>
<tr>
<td>mov_ob2ear OB_i_j EAR_k</td>
<td><code>EAR[k] = OB[i].OB[j]</code></td>
</tr>
<tr>
<td>mov_ob2reg OB_i UDPR_j</td>
<td><code>UDPR[j] = OB[i]</code></td>
</tr>
<tr>
<td>yield (opsize)</td>
<td>Suspends current thread execution, releases OB registers corresponding to current event being processed</td>
</tr>
<tr>
<td>yield_terminate (opsize)</td>
<td>Terminates current thread execution, releases OB and UDP registers corresponding to current event and thread being processed resp</td>
</tr>
<tr>
<td>send_with_ret Re, Rw, Rd,$sz</td>
<td>add lane_ID, TID of caller to message Re - event_word that message return should trigger Rw - data to be sent Rd - destination lane $sz - Number of bytes of Rw</td>
</tr>
<tr>
<td>send_reply Re, Rw,$sz</td>
<td>Use source info (lane_ID, PTID) sent using send_with_ret for sending response Re - event_word that message response should trigger Rw - data word/value to be sent back $sz - Number of bytes of Rw</td>
</tr>
<tr>
<td>ev_update_1 Rs, Rd, $imm1, $mask</td>
<td>Rs - source event_word Rd - destination event_word $imm - immediate value of field to be updated $mask - 4 bit mask to indicate which field is to be updated in event_word</td>
</tr>
<tr>
<td>ev_update_2 Rd, $imm1, $imm2, $mask</td>
<td>Rd - destination event_word $imm1 - immediate value of field 1 to be updated $imm2 - immediate value of field 2 to be updated $mask - 4 bit mask to indicate which fields are to be updated in event_word</td>
</tr>
</tbody>
</table>
New UpStream Actions

This section describes the new actions added to UDP ISA for Upstream in detail.

Conditional Branch instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bne R1, R2, target</td>
<td>Branch to target if R1 != R2. R1, R2 are registers in OB</td>
</tr>
<tr>
<td>beq R1, R2, target</td>
<td>Branch to target if R1 == R2. R1, R2 are registers in OB</td>
</tr>
<tr>
<td>bgt R1, R2, target</td>
<td>Branch to target if R1 &gt; R2. R1, R2 are registers in OB</td>
</tr>
<tr>
<td>blt R1, R2, target</td>
<td>Branch to target if R1 &lt; R2. R1, R2 are registers in OB</td>
</tr>
</tbody>
</table>

Condition Branches allow conditional control flow based on registers. The target to be taken is an action within the current activation or a shared action block.

Example usage:
- bgt UDPR_0 UDPR_1 l1 - jump to action at ‘l1’ if UDPR_0 > UDPR_1
- bne OB_0 UDPR_2 block_1 - jump to shared block ‘block_1’ if OB_0 != UDPR_2

Instructions for Fast Messaging

The send instruction allows for unified messaging and data transfer between TOP cores, Upstream lanes and DRAM.

Send Messages for read data:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>send Re, Rd, Ra,$sz, $rw, md</td>
<td>Re - event_word that message return should trigger Rd - destination ID Ra - Register for Addressing $sz - Number of bytes to read/write (imm value) $rw - read / write bit (imm value) md - addressing mode (all values for Re, Rd, Ra could come from software thread state -- with limits for encoding)</td>
</tr>
</tbody>
</table>
Fig 7: Message construction using send message for reads

- **DestinationID**: Destination nodes for the message. Destination IDs are assigned based on the participating nodes in the system. For a system with a 64 lane UDP, TOP (n cores) and DRAM, following could be possible assignment. (This assignment needs to be created at system startup and stored in a shared location for lookup?)

<table>
<thead>
<tr>
<th>Node</th>
<th>Dest ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRAM</td>
<td>0</td>
</tr>
<tr>
<td>TOP</td>
<td>1 … n</td>
</tr>
<tr>
<td>UpStream Lanes</td>
<td>n+1 … n+64</td>
</tr>
</tbody>
</table>

- **md - Addressing mode**: Addressing mode decides the usage of the address register Ra in the instruction. For read messages, the mode refers to using Ra either as the address of Local Memory or using Ra as the offset to a global base address in EAR (0|1|2|3).

<table>
<thead>
<tr>
<th>Mode ($rw = r$)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ra - Address of location in LM</td>
</tr>
<tr>
<td>1</td>
<td>Ra - Offset of global address from base EAR0</td>
</tr>
<tr>
<td></td>
<td>Global Addr = EAR0+Ra</td>
</tr>
</tbody>
</table>
|   | Ra - Offset of global address from base EAR1  
Global Addr = EAR1+Ra |
|---|---|
| 3 | Ra - Offset of global address from base EAR2  
Global Addr = EAR2+Ra |
| 4 | Ra - Offset of global address from base EAR3  
Global Addr = EAR3+Ra |

- Ra - Register for addressing usage differs based on addressing mode.
  - For Read messages, In LocalMemory mode, Ra gives the 32bit address of the Local Memory location directly. In DRAMMemory mode, EAR+Ra gives the 64-bit DRAM memory location.

Send Messages for write data:

For write messages, the destination can either be DRAM memory, scratch pads of other UpStream nodes or operand buffers / event queues of an Upstream lane (including the same lane). These two cases are distinguished by the address mode and mentioned below.

| send Re, Rw, Ra,$sz, $rw, md | Re - event_word that message return should trigger  
Rw - Data word  
Ra - Register for Addressing (destination address)  
$sz - Number of bytes to read/write (imm value)  
$rw - read / write bit (imm value)  
md - addressing mode  
(all values for Re, Rd, Ra could come from software thread state -- with limits for encoding) |

All the operands except Rw carry the same meaning as in the read messages.
- Rw - Register containing the value of data to be written or the address in LM containing the value of the data to be written
Fig 8: Message construction using send message for writes

Modes in Write Messages

<table>
<thead>
<tr>
<th>Mode ($rw = w$)</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0              | Ra - Destination ID within node  
                Rw - data to be written |
| 1              | Ra - Offset for global address calculation  
                Global Addr = EAR0+Ra  
                Rw - data to be written |
| 2              | Ra - Offset for global address calculation  
                Global Addr = EAR1+Ra  
                Rw - data to be written |
| 3              | Ra - Offset for global address calculation  
                Global Addr = EAR2+Ra  
                Rw - data to be written |
| 4              | Ra - Offset for global address calculation  
                Global Addr = EAR3+Ra  
                Rw - data to be written |
| 5              | Ra - Destination ID within node  
                Rw - LM address of data to be written |
| 6              | Ra - Offset for global address calculation  
                Global Addr = EAR0+Ra  
                Rw - LM address of data to be written |
Ra - Offset for global address calculation
Global Addr = EAR1+Ra
Rw - LM address of data to be written

Ra - Offset for global address calculation
Global Addr = EAR2+Ra
Rw - LM address of data to be written

Ra - Offset for global address calculation
Global Addr = EAR3+Ra
Rw - LM address of data to be written

Note: LM is not addressed since “mov_reg2lm” can be used for writing into LM

Examples:

**Data streaming (different data pointer)**
Send udpr_1, udpr_2, OB_1,#16,1
Add udpr_3, OB_1, #16
Send udpr_1, udpr_2, udpr_3,#16,1
Add udpr_3, udpr_3, #16
Send udpr_1, udpr_2, udpr_3,#16,1

**Fan-out messages (different destinations)**
Send udpr_1, udpr_2, OB_1,#4,1
Add udpr_2, udpr_2, #lane_increment
Send udpr_1, udpr_2, OB_1,#4,1
Add udpr_2, udpr_2, #lane_increment
Send udpr_1, udpr_2, OB_1,#4,1

Send messages with source information:

send_with_ret messages add source information to the messages. This allows for inter-thread communication between UpStream lanes. Typical use case for send_with_ret message is in creating threads to execute functions either on the same lane or on a different lane. The addition of source information to the message allows for the callee thread to return to the caller thread on completion of thread execution. In this case, the parameters and addressing mode are similar to the send messages for write data, since typically data passed on to the callee thread needs to be written into the operand buffer of the lane on which it will be executed.
send_with_ret Re, Rw, Rd,$sz

- Re - event_word that message return should trigger
- Rw - data to be sent
- Rd - destination lane
- $sz - Number of bytes of Rw

(all values for Re, Rw, Rd could come from software thread state), Lane ID and Thread ID are appended to the message as operands.

Fig 9: Message construction using send message for with source information

Send messages using return information:

send_reply Re, Rw, $sz

- Re - event_word that message return should trigger
- Rw - data word/value to be sent back
- $sz - Number of bytes of Rw

(all values for Re, Rw could come from software thread state)

send_reply messages allow the usage of operands sent specifically with send_with_ret messages directly to decide dest ID and thread ID. The messages are constructed as shown below.
Event Update instructions

The following instructions can be used to create/format new event words efficiently when one to two fields of the event word are to be modified. `ev_update_1` instruction clearly specifies the source register for the event word whereas the source register in `ev_update_2` is implicit and is always EvQ.Top.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
</table>
| `ev_update_1 Rs, Rd, $imm1, $mask` | Rs - source event_word  
Rd - destination event_word  
$imm1 - immediate value of field 1 to be updated  
$mask - 4 bit mask to indicate which field is to be updated in event_word |
| `ev_update_2 Rd, $imm1, $imm2, $mask` | Rd - destination event_word  
$imm1 - immediate value of field 1 to be updated  
$imm2 - immediate value of field 2 to be updated  
$mask - 4 bit mask to indicate which fields are to be updated in event_word |
| `ev_update_reg_2 Rs, Rd, R1, R2, $mask` | Third possibility similar to `ev_update_2` with values coming from regs. |
Synchronization instructions

| cmpswp reg, mem, imm1, imm2 | mem = imm2 if mem == imm1 (reg = mem) |

Enables synchronization across upstream lanes. cmpswp compares the value in mem (shared address in LM), and sets mem to imm2 if mem is equal to imm1. It returns the existing value of mem in reg. This can be used to implement locks.

Example: UpStream Kernel to implement a lock using cmpswp instruction.

l1:  mov_imm2reg UDPR_0 <lock_addr>
    cmpswp UDPR_1, UDPR_0, $0, $1
    bne UDPR_1, $0 l1
    … <lock_acquired> …

Additional Move Instructions

<table>
<thead>
<tr>
<th>mov_ob2ear OB_i_j EAR_k</th>
<th>EAR[k] = OB[i].OB[j]</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov_ob2reg OB_i UDPR_j</td>
<td>UDPR[j] = OB[i]</td>
</tr>
</tbody>
</table>

mov_ob2ear moves values from a pair of operand buffer registers (i,j) to the 64 bit extended address register.
mov_ob2reg moves values from the operand buffer register to the general purpose UDP registers.

Thread Switch and Terminate Instructions

<table>
<thead>
<tr>
<th>yield (opsize)</th>
<th>Suspends current thread execution, releases OB registers corresponding to current event being processed</th>
</tr>
</thead>
<tbody>
<tr>
<td>yield_terminate (opsize, regsize)</td>
<td>Terminates current thread execution, releases OB and UDP registers corresponding to current event and thread being processed resp</td>
</tr>
</tbody>
</table>
UpStream Thread Lifecycle

The Upstream thread is a lightweight execution abstraction that includes the thread state defined under UpStream SW Thread state.

### Fig 11: Thread State Table

<table>
<thead>
<tr>
<th>ID</th>
<th>regbase</th>
<th>regsiz</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>3</td>
</tr>
</tbody>
</table>

**Thread Creation and thread state table**

A sent instruction with an invalid_thread_ID (FF) within the event word, creates a new thread, and the hardware allocates it a unique thread ID. This is inserted into the live threads table called the thread state table. The thread state table is used for thread management. GPR is allocated for the kernel, and regbase is stored in the live thread table. Sufficient registers must be available in the GPR for the thread to be created and launched.
Thread Switching:
UpStream threads can switch on event boundaries. Currently threads cannot be preempted. Threads are created and triggered into execution by events. On yield, the current thread’s event releases the operand buffer slots, but retains the GPR which allows the thread to resume execution on future events in a single cycle. The thread ID in an event helps activating the right thread from the thread state table and setting the regbase to use the corresponding set of registers for the thread.

Thread Termination:
Thread termination is effected on executing a yield_terminate instruction. The yield_terminate instruction releases both the operand buffer slots corresponding to the thread’s current event and the GPR registers that it currently holds as thread state. The thread is removed from the thread state table as well.

(Upstream Management API)
UpStream Lane MicroArchitecture

The enhancements described above are implemented as an extension of the UDP lane microarchitecture and a schematic of this microarchitecture is shown in the Fig below.

The Network Interface is split into Incoming and Outgoing portions for clarity of data flow. The incoming network interface unpacks messages into the Event Queue and Operand Buffer. The Event Queue feeds the EventQ.Top to the Event Dispatch block. The Event Dispatch block decides the codelet to be executed and subsequently the codelet instructions are fetched in the IF stage. The Event also contains the TID that fetches the current Thread state from the Thread State table. This gives the window of registers in the register file that are valid for the current thread. Finally the Operands in the operand buffer are fed into the Ex unit to be used as operands in the actions being executed.

The outgoing Network Interface is fed by message parameters from the send instruction which also feeds the AGU that calculates the DRAM address and based on the addressing mode, the right address is used to pack the message in the Network interface before being sent out.

Fig 13: UpStream Lane MicroArchitecture.

UpStream Lane adds 5 elements (orange) to the UDP lane (green) which builds on a traditional CPU pipeline (grey)
System Memory Map

The Upstream SRAM is integrated into the Global Address Space above DRAM addresses into the PCI/PCIe Memory Range. Writing into this address range effectively results in writing into the UpStream SRAM.

![Sample x86-64 Memory Map](image1)

The scratch pad memory itself is interleaved depending on the size of the memory. The 64-bit Physical Address can be segmented as shown below.

![64-bit Physical Address bit-mapping](image2)

Based on the above bit-mapping the interleaving in the Scratch Pad is shown in the figure below.
Fig 16: Physical Address interleaving across LM banks

For larger sizes of the scratch pad, more blocks are added per lane and bits are successively added beyond 21 to address the added blocks. A sample address map for a 4MB scratch pad is shown in the table below with UP_BASE pointing to the base address of the scratch pad.

<table>
<thead>
<tr>
<th>Start_addr</th>
<th>End_addr</th>
<th>Size</th>
<th>SRAM Block Identification</th>
</tr>
</thead>
<tbody>
<tr>
<td>UP_BASE+00-0000</td>
<td>UP_BASE+00-0FFF</td>
<td>4K</td>
<td>Bank0/Block0</td>
</tr>
<tr>
<td>UP_BASE+00-1000</td>
<td>UP_BASE+00-1FFF</td>
<td>4K</td>
<td>Bank1/Block0</td>
</tr>
<tr>
<td>UP_BASE+00-2000</td>
<td>UP_BASE+00-2FFF</td>
<td>4K</td>
<td>Bank2/Block0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>4K</td>
<td>...</td>
</tr>
<tr>
<td>UP_BASE+03-F000</td>
<td>UP_BASE+03-FFFF</td>
<td>4K</td>
<td>Bank63/Block1</td>
</tr>
<tr>
<td>UP_BASE+04-0000</td>
<td>UP_BASE+04-0FFF</td>
<td>4K</td>
<td>Bank0/Block1</td>
</tr>
<tr>
<td>UP_BASE+04-1000</td>
<td>UP_BASE+04-1FFF</td>
<td>4K</td>
<td>Bank1/Block1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>4K</td>
<td>...</td>
</tr>
<tr>
<td>UP_BASE+3C-0000</td>
<td>UP_BASE+3C-0FFF</td>
<td>4K</td>
<td>Bank0/Block15</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>4K</td>
<td>...</td>
</tr>
<tr>
<td>UP_BASE+3F-F000</td>
<td>UP_BASE+3F-FFFF</td>
<td>4K</td>
<td>Bank63/Block15</td>
</tr>
</tbody>
</table>

Table: Address Mapping with 4k interleaved blocks in UpStream ScratchPad (size=4M)
UpStream Examples

Example 1: Strided Address Generation

This example highlights the use of a single UpStream lane in generating a stream of strided addresses. The UpStream kernel loop for this and an illustrative timing diagram to show the output stream of references is shown below.

UpStream Kernel (4B strided addresses)

```
li:    addi UDPR_0, UDPR_0, 4       (+4)
       send Re, Rd, UDPR_0, sz, r, 1  (s)
       bne l1, $32, l1               (b)
```

The above kernel generates 1 reference in 3 instructions giving 0.33 references per instruction per lane. That translates to a total of ~21B refs/s for 64 1GHz UpStream lanes.
Example2: Tail Recursion using threads

In this example we highlight the creation of light-weight threads and their usage in implementing function calls and returns on the UpStream lane. This example recursively calculates \( \text{sum}(n) \) (sum of \( n \) natural numbers). The Top processor triggers the first thread on an UpStream lane with event “#rejsonsum” and \( n \) as an operand in OB_0.

PseudoCode for example2:

\[
\text{sum}(n):
\begin{align*}
\text{If } (n < 2) \\
\quad & \text{Return 1} \\
\text{else} \\
\quad & \text{Return } n + \text{sum}(n-1)
\end{align*}
\]

Event_word from top - [lane_id(up_1),FF, base0, #rejsonsum]

Fig: Tail Recursion call/return sketch in UpStream kernel

UpStream Kernel

\[
\begin{align*}
\text{#rejsonsum} \\
\text{mov}_\text{ob2reg OB}_0 \text{UDPR}_1 & \rightarrow \text{save } n \\
\text{blt } \text{UDPR}_1, S2, l1 & \rightarrow \text{if } n < 2 \text{ go to } l1 \\
\text{subi } \text{UDPR}_1, S1, \text{UDPR}_3 & \rightarrow n-1 \\
\text{send}_\text{with}_\text{ret} \text{EQT, UDPR}_3, \text{up}_1, S4 & \rightarrow \text{call for } n-1 \\
\text{yield}(1) \\
\text{l1: } \text{mov}_\text{imm2reg} \text{UDPR}_2, <\#\text{returnsum}> & \rightarrow \text{create new event_word for } \#\text{returnsum} \\
\text{send}_\text{reply} \text{UDPR}_2, S1, S4 & \rightarrow \text{send } \#\text{returnsum} \text{ event_word} \\
\text{yield}_\text{terminate}(1, 3) & \rightarrow \text{terminate and release operands+GPR} \\
\text{#returnsum} \\
\text{add } \text{OB}_0, \text{UDPR}_1, \text{UDPR}_4 & \rightarrow n+\text{sum}(n-1) \\
\text{beq } \text{UDPR}_1, \text{#n}, l2 & \rightarrow \text{check if this is thread udpr}_1 = n \\
\text{send}_\text{reply} \text{EQT, UDPR}_4, \text{#4} & \rightarrow \text{send } \#\text{returnsum} \text{ event_word with sum} \\
\text{yield}_\text{terminate}(1, 4) & \rightarrow \text{terminate and release operands+GPR} \\
L2: \text{send } \text{UDPR}_2, \text{UDPR}_4, \text{Ra?}, \text{#4}, \text{w,md?} & \rightarrow \text{send final sum to TOP} \\
\text{yield}_\text{terminate}(1, 4) & \rightarrow \text{Terminate and release operands}
\end{align*}
\]

Performance Analysis:

The above kernel performs

1. A calling sequence in 3 instructions (1 arg, + 1 inst for every additional arg)
2. A return sequence in 3 instructions (4 inst including work in this example)

This is as opposed to 2 and 1 in a conventional processor resp.
Example 3: Tail Recursion using threads on successive lanes

This example illustrates example 2 being done on successive lanes. With the addition of 2 extra instructions, the threads can be called on successive lanes and returned back to the original lane.

Fig: Tail Recursion call/return sketch in UpStream kernel on successive lanes

UpStream kernel

```
#reqsum
    mov_ob2reg OB_0 UDPR_1        → save n
    blt UDPR_1, $2, l1           → if n < 2 go to l1
    subi UDPR_1, $1, UDPR_3      → n-1
    rshift_and_imm EQT, UDPR_5, 24, 0xff → get prev lane ID
    addi UDPR_5 UDPR_5 $1       → increment lane num
    send_with_ret EQT, UDPR_3, UDPR_5, $4 → call for n-1
    yield(1)

l1:  mov_imm2reg UDPR_2,#returnsum> → create new event_word for #returnsum
    send_reply UDPR_2, up_1, $1, 4, w) → send #returnsum event_word
    yield_terminate (1, 3) → terminate and release operands+GPR

#returnsum
    add OB_0, UDPR_1, UDPR_4      → n+sum(n-1)
    beq UDPR_1, #n, l2           → check if this is thread udpr_1 = n
    send_reply EQT, UDPR_4, #4    → send #returnsum event_word with sum
    yield_terminate(1, 4) → terminate and release operands+GPR

L2:   send UDPR_2, UDPR_4, #4   → send final sum to TOP
    yield_terminate(1, 4) → Terminate and release operands
```
Example 4: Fast Messaging - Stream DRAM data into ScratchPad

This example demonstrates the use for Fast messaging to stream DRAM data into UpStream ScratchPad. The Upstream lane is provided with the DRAM base address (OB_0_1), LM base address (OB_2) and total number of bytes to be transferred (OB_3) by Top and (#fetch) event is triggered.

UpStream Kernel

```
#fetch
    mov_ob2reg OB_0_1 EAR0    \rightarrow DRAM base address
    mov_ob2reg OB_2 UDPR_1    \rightarrow LM base address
    mov_ob2reg OB_3 UDPR_2    \rightarrow total bytes
    mov_imm2reg $0 UDPR_0     \rightarrow initialize offset
11: send <#retdata>, up_1, UDPR_0, $64, r, 1 \rightarrow fetch 64 bytes (cache-line size)
    addi UDPR_0, $64, UDPR_0  \rightarrow update offset
    ble UDPR_0 UDPR_2 11     \rightarrow offset > size
    yield(1)
#redata
    copy OB_0 UDPR_1 $64     \rightarrow move word[0]
    addi UDPR_1, UDPR_1 $64  \rightarrow update LM addr
    ble UDPR_1 UDPR_2 12     \rightarrow all bytes received
    yield(64)
12: yield_terminate(64)
```

Performance Analysis: The kernel uses 3 instructions to generate back to back fetch requests (as in the strided case in example 1), but in this case the fetch size is set to 64-bytes. In the return path, 4 instructions are required to store 64B i.e 16B/lane/cycle giving ~1.024TB/s for 64 lanes.
Example 5: Fast Messaging - Stream Scratch Pad data into DRAM

This example illustrates the use for Fast messaging to write back Scratch Pad data into DRAM. The Upstream lane is provided with the DRAM base address (OB_0_1), LM base address (OB_2) and total number of bytes to be transferred (OB_3) by Top and (#fetch) event is triggered.

UpStream Kernel:

```
#write-back
    mov_ob2reg OB_0_1 EAR0 → DRAM base address
    mov_ob2reg OB_2 UDPR_1 → LM base address
    mov_ob2reg OB_3 UDPR_2 → total bytes
    mov_imm2reg $0 UDPR_0 → initialize offset

11: send <ack> up_1, UDPR_1, UDPR_0, $64, w, 1 → write 64 bytes (data word)
    addi UDPR_0, $64, UDPR_0 → update offset for DRAM
    addi UDPR_1, $64, UDPR_1 → update LM Addr
    ble UDPR_0 UDPR_2 l1 → offset > size
    mov_imm2reg $0 UDPR_0
    yield(4)

#ack
    addi UDPR_0 $64 UDPR_0
    bge UDPR_0 UDPR_2 l2 → offset > size
    yield(0)

l2: yield_terminate(1,2)
```

Performance Analysis: The kernel uses 4 instructions to generate back to back write-back messages with write-back size set to 64-bytes giving ~1.024TB/s write BW for 64 lanes.
Example 6: Binary Recursion

The below code is an example of an Upstream kernel that returns the fibonacci of n using binary recursion.

**Functionality: (Fibonacci)**

```plaintext
fib(n):
    if (n < 3)
        return 1;
    return fib(n-1) + fib(n-2);
```

**Event word from Top:**
event_word - [up_1, FFFF, base0, #fib]

**Upstream Program:**

```assembly
mov_ob2reg OB_0 UDPR_1  → UDPR1= n
mov_reg2reg EQT, UDPR_2  → use event_word as is with invalid TID
blt UDPR_1, $3, L1     → if n <3 go to L1 [fib(1) and fib(2)]
mov_reg2reg UDPR1 UDPR3  → UDPR3 used to show n-1 and n-2
L3:  subi UDPR_3, $1, UDPR_3  → UDPR3= n-(i*1)
    send UDPR_2, up_1, UDPR_3, $4, w,0  → call fib( n-(i*1) ) [create a new thread]
    blt UDPR_3 2 L3
    yield(1)

#ret
add OB_0 UDPR_4 UDPR_4  → UDPR4=returned val+UDPR4
subi UDP1, UDPR5, 2     → UDPR5=n-2
bnq UDP3, UDP5, L3       → UDPR5=n-2
beq UDPR_1, #n, L2      → goto L2 [send the result to top]
mov_imm2reg UDPR_2,#ret  → event label = #ret
rshift_and_imm UDPR_5, TS, $8, $00ff0000  → for embedding PTID into event label
bitwise_or UDPR_2, UDPR_5, UDPR_2  → embed PTID in event_word
send UDPR_2, up_1, UDPR_4, 4, w,0  → send addition result to parent thread
yield_terminate(1,4)

L1:  mov_imm2reg UDPR_2,#ret  → event label = #ret
    rshift_and_imm UDPR_5, TS, $8, $00ff0000  → for embedding PTID into event label
    bitwise_or UDPR_2, UDPR_5, UDPR_2  → embed PTID in event_word
    send UDPR_2, up_1, $1, 4, w,0  → send $1 to parent thread [fib(3)]
yield_terminate (1,4)  → terminate and release operands+GPR

L2:  send UDPR_2, top, UDPR_4, #4, w,0  → event label to top=event label from top
     yield_terminate(1,4)
```
Example 7: fork/join

Functionality:

begin(a, b):

For (i=0; i<2; i++)
    If (i=a)
        new_a = a+a
    If (i=b)
        new_b = b+b

c = new_a + new_b
Send c to the top

Upstream kernel:

#begin
    // read first argument
    mov_ob2reg OB_1 UDPR_1                       → save argument a
    mov_ob2reg OB_2 UDPR_3                       → save argument b
    mov_reg2reg EQT, UDPR_2                      → use event_word as is with invalid TID
    send(#join-create, up_1, XX, $4, w,0)       → create join with arbitrary Operand
    // yield to wait for join thread’s TID
    yield(1)
#joincreated
    // read the join TID
    mov_ob2reg UDPR_4                           → read join TID
    // create first fork with a
    send(#fork, up_2, UDPR_4, $4, w,0)          → create first fork with joinTID
    send(#arg2, up_2, UDPR_1, $4, w,0)          → send a to first fork
    send(#fork, up_3, UDPR_4, $4, w,0)          → create second fork with joinTID
    send(#arg2, up_3, UDPR_3, $4, w,0)          → send b to second fork
    yield_terminate (1,x)
#fork
    // read destination TID
    mov_ob2reg UDPR_1                           → save DTID
    shifll UDPR_1 UDPR_1 8                      → move DTID to TID location in event word
    yield(1)
    // read a | b
#arg2 mov_ob2reg UDPR_3                       → save a | b
    add UDPR_3 UDPR_3 UDPR_3                    → UDPR3=a+a
//create event word
mov_imm2reg UDPR_2,#forkarrive → event label = #forkarrive
bitwise_or UDPR_2, UDPR_1, UDPR_2 → embed DTID in event_word
send(UDPR_2, up_1,UDPR_3, $4, w,0) → send res to join
yield_terminate (1,x)

#join-create
//get TS info
rshift_and_imm UDPR_1, TS, $8, $00ff0000 → fextract TID
//create event word
mov_imm2reg UDPR_2,#joincreated → event label = #joincreated
rshift_and_imm UDPR_3, TS, $8, $00ff0000 → for embedding PTID into event label
bitwise_or UDPR_2, UDPR_3, UDPR_2 → embed PTID in event_word
send(UDPR_2, up_0,UDPR_1, $4, w,0) →send TID to parent

#fork-arrive
L1: yield(1)
Addi UDPR_6, $1, UDPR_6
add OB_0 UDPR_4 UDPR_4
bnq UDPR_6 $2 L1
send (UDPR_2, top, UDPR_4, #4, w,0) → event label to top=event label from top
yield_terminate (1,x)

Issue1: From Fork-Join

1. Is there a way to signal successive events belong to the same thread before thread is created.
   [Right now answer is No.]

With send_w_ret and send_reply

#begin
//read first argument
mov_ob2reg OB_1 UDPR_1 → save argument a
mov_ob2reg OB_2 UDPR_3 → save argument b
mov_reg2reg EQT, UDPR_2 → use event_word as is with invalid TID
send_with_ret #joincreated, XX, up_1, $4 →create the join thread
yield(1) → yield to wait for join thread’s TID
#joincreated
mov_ob2reg UDPR_4 → read join TID
send(#fork, up_2, UDPR_4, $4, w,0) → create first fork with joinTID
send(#arg2, up_2, UDPR_1, $4, w,0) → send a to first fork
send(#fork, up_3, UDPR_4, $4, w,0) → create second fork with joinTID
send(#arg2, up_3, UDPR_3, $4, w,0) → send b to second fork
yield_terminate (1,x)

#fork
mov_ob2reg UDPR_1 → read and save DTID
shiftl UDPR_1 UDPR_1 8 → move DTID to TID location in event word
yield(1)

#arg2
mov_ob2reg UDPR_3 → read and save a | b
add UDPR_3 UDPR_3 UDPR_3 → UDPR3=a+a
mov_imm2reg UDPR_2,#forkarrive → event label = #forkarrive
bitwise_or UDPR_2, UDPR_1, UDPR_2 → embed DTID in event_word
send(UDPR_2, up_1,UDPR_3, $4, w,0) → send res to join
yield_terminate (1,x)

#join-create
rshift_and_imm UDPR_1, TS, $8, $00ff0000 → extract TID
mov_imm2reg UDPR_2,#joincreated → event label = #joincreated
send_reply UDPR_2, UDPR_1, $4 → send TID to parent

#fork-arrive
L1: yield(1)
    Addi UDPR_6, $1, UDPR_6
    add OB_0 UDPR_4 UDPR_4
    bnq UDPR_6 $2 L1
    send (UDPR_2, top, UDPR_4, #4, w,0) → event label to top=event label from top

With event_update instructions

#begin
//read first argument
mov_ob2reg OB_1 UDPR_1 → save argument a
mov_ob2reg OB_2 UDPR_3 → save argument b
mov_reg2reg EQT, UDPR_2 → use event_word as is with invalid TID
send_with_ret #joincreated, XX, up_1, $4 //create the join thread
yield(1) // yield to wait for join thread’s TID

#joincreated
mov_ob2reg UDPR_4 → save join TID
send(#fork, up_2, UDPR_4, $4, w,0) → create first fork with joinTID
send(#arg2, up_2, UDPR_1, $4, w,0) → send a to first fork
send(#fork, up_3, UDPR_4, $4, w,0) → create second fork with joinTID
send(#arg2, up_3, UDPR_3, $4, w,0) → send b to second fork
yield_terminate (1,x)

#fork
mov_ob2reg UDPR_1 → read and save DTID
shiftl UDPR_1 UDPR_1 8  \rightarrow move DTID to TID location in event word

yield(1)

#arg2  mov_ob2reg UDPR_3  \rightarrow read and save a | b

add UDPR_3 UDPR_3 UDPR_3  \rightarrow UDPR3=a+a

ev_update_1 UDPR_2, UDPR_1, <forkarrive>, 1  \rightarrow one instruction reduced

send(UDPR_2, up_1, UDPR_3, $4, w,0) \rightarrow send res to join

yield_terminate (1,x)

#join-create

rshift_and_imm UDPR_1, TS, $8, $00ff0000  \rightarrow extract TID

mov_imm2reg UDPR_2,<#joincreated>  \rightarrow event label = #joincreated

send_reply  UDPR_2, UDPR_1, $4  \rightarrow send TID to parent

#fork-arrive

L1:  yield(1)

Addi UDPR_6, $1, UDPR_6

add OB_0 UDPR_4 UDPR_4

bnq UDPR_6 $2 L1

send (UDPR_2, top, UDPR_4, #4, w,0) \rightarrow event label to top=event label from top

Ev_update_2 Rs, Rd, R1, R2 $mask

ev_update_2 UDPR_
Example 8: Triangle Counting

The below code is a complete example of an Upstream kernel for a graph processing workload that calculates the triangle count in a given graph.

```
#vr
  lshift_and_imm OB_0 UDPR_1 2 4294967295        #0 v1 size
  mov_ob2reg OB_2 UDPR_4                                     #1 v1 - LM base
  lshift_and_imm OB_4 UDPR_2 2 4294967295        #2 v2 size
  mov_ob2ear OB_6_7 EAR_0                                    #3 v2 DRAM base
  mov_imm2reg UDPR_3 0                                      #4 TC / iterator
  mov_imm2reg UDPR_5 0                                      #5 v1 index
  mov_imm2reg UDPR_6 0                                      #6 v2 index
  rshift_and_imm TS UDPR_7 0 16711680                 #7 Extract TID for event_word
  addi UDPR_7 UDPR_7 " + str(event_map["nr"]))    #8 UDPR_9 has event_word for n1
  mov_imm2reg UDPR_11 2" ) # Lane 1                #9

# Fetch phase
  beq UDPR_2 UDPR_3 #14                                       #10  11
  send UDPR_7 UDPR_11 UDPR_3 4 r 1                   #11  UDPR7 - event_word, UDPR_11 - Lane ID,
                                         UDPR_3 - addr_offset, 4 - sz, r - read, 1 -
                                         addr_mode (EAR0)
  addi UDPR_3 UDPR_3 4                                          #12
  jmp #10                                                                      #13
  mov_imm2reg UDPR_3 0                                      #14 y1
  mov_imm2reg UDPR_9 0                                      #15 walk = 0
  add UDPR_4 UDPR_5 UDPR_7                              #16 lm addr calculation
  mov_lm2reg UDPR_7 UDPR_8 4                            #17 fetch v1 from lm
  yield 2                                                                        #18

#nr
  # Pick up first element
  ble UDPR_8 OB_0 #3                                               #0 if n1 > n2 set walk -1
  mov_imm2reg UDPR_9 -1                                     #1
  jmp #4                                                                      #2
  mov_imm2reg UDPR_9 1                                      #3
  blt UDPR_9 0 #16                                                    #4 +1 walk

  # walk +1 (forward) [loop]
  bne UDPR_8 OB_0 #8                                          #5
  addi UDPR_3 UDPR_3 1                                     #6
  jmp #27                                                                      #7 block 1 - term/yield block
  bgt UDPR_8 OB_0 #27                                     #8
```
addi UDPR_5 UDPR_5 4 #9
bne UDPR_5 UDPR_1 #13 #10
subi UDPR_5 UDPR_5 4 #11
jmp #27 #12
addi UDPR_4 UDPR_5 UDPR_7 #13 lm addr calculation
mov_lm2reg UDPR_7 UDPR_8 4 #14 fetch v1 from lm
jmp #5 #15

#walk -1 (backward)
bne UDPR_8 OB_0 #19 #16
addi UDPR_3 UDPR_3 1 #17
jmp #27 #18 block 1 - term/yield block
blt UDPR_8 OB_0 #27 #19
subi UDPR_5 UDPR_5 4 #20
bge UDPR_5 0 #24 #21 #y2
addi UDPR_5 UDPR_5 4 #22
jmp #27 #23
addi UDPR_4 UDPR_5 UDPR_7 #24 lm addr calculation
mov_lm2reg UDPR_7 UDPR_8 4 #25 fetch v1 from lm
jmp #16 #26

#<next v2>
addi UDPR_6 UDPR_6 4 #27 #y1
beq UDPR_6 UDPR_2 #30 #28
yield 1 #29
send tc TOP UDPR_3 4 w 0 #30 #fin
yield_terminate 4 #31
References

[1]. Yuanwei Fang, Andrew A. Chien. “UDP System Interface and Lane ISA Definition” (Tech report details to be added)

To Add:

Instructions:

Mov_lm2reg_blk
block_compare
block_compare_i
mask_or