UDP System Interface and Lane ISA Definition

Yuanwei Fang and Andrew A. Chien

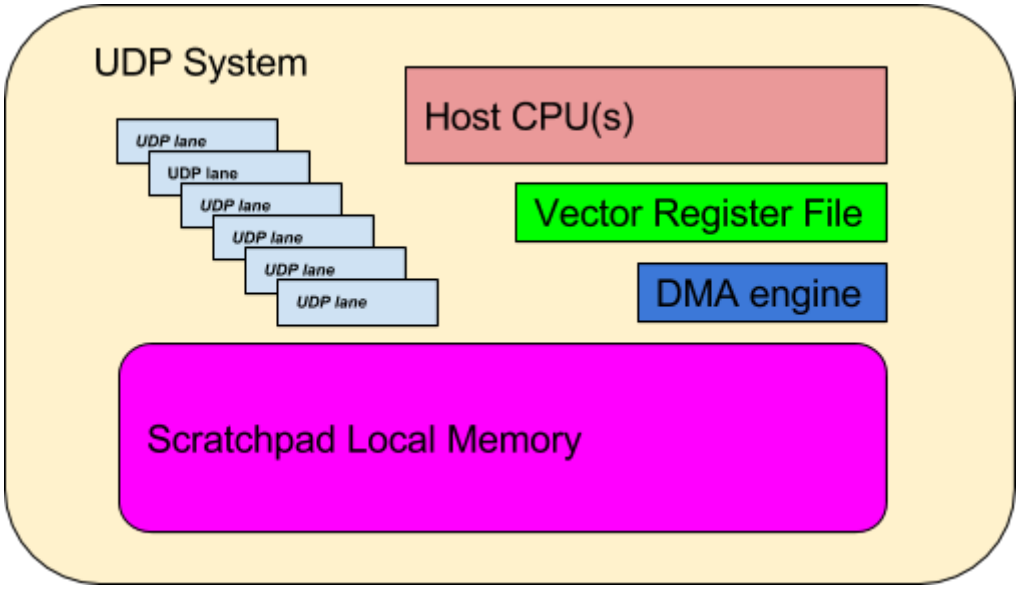
Table of Contents:

1. Preface

This document contains the Unstructured Data Processor (UDP) hardware architecture and tool-chain software architecture description. It contains the UDP assembly-code instruction, machine-code instruction and how assembly-code translates to the machine code. In addition, implementation of UDP ISA in the hardware is also described. High-level related mechanisms have also been published in the academic literature [1,2,3].

2. UDP System Overview

The UDP is an MIMD parallel accelerator with each lane generating memory accesses, and the 64-lanes collectively sharing a multi-bank local memory. The building block of the UDP system is the UDP lane. Each lane contains 16 general-purpose scalar data registers and a stream buffer equipped with automatic indexing management and streams prefetching logic. Generally, a UDP system consists of: a host CPU, UDP lanes, vector register file, scratch on-chip memory, and a DMA engine. The figure below shows the system.



Functionality of each component

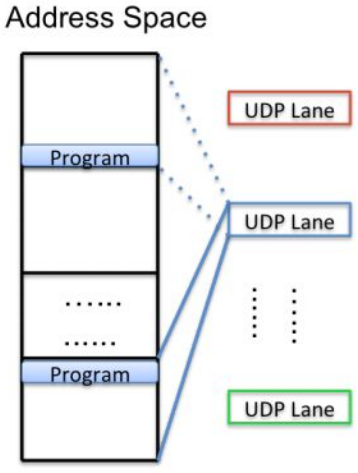
- Host CPU: general code execution, memory management, UDP context-switch, exception handling, etc.

- Vector Register File: a shared 64x2048-bit vector register file. It provides the data value for parallel UDP lanes.
- UDP lane: building block of the UDP system. UDP program execution engine unit.
- Scratchpad local memory (Local Memory): store UDP program, UDP general data storage, and temporary buffer to provide massive off-chip bandwidth.
- DMA engine: background data movement engine transferring data between main memory and the local memory.

3. UDP System Memory Model

In this section, we describe the memory semantics that is specified by the UDP system architecture.

The UDP is an MIMD parallel accelerator with each lane generating memory accesses, and the 64-lanes collectively sharing a multi-bank local memory. It adopts "Restricted Addressing" scheme, which is a hybrid scheme. Restricted addressing adds a base register to each UDP lane. This base allows code generation similar to that with local addressing. To shift the addressable window, the UDP lane changes its base register value under software control. With compiler support, a UDP lane can access full local memory address (see figure below).



Once UDP lanes can concurrently address the same memory location (global or flexible), memory consistency issues arise. UDP lane programs are all generated by a single compiler (no multiprogramming) and operate nearly synchronously, so lane interaction can be managed and minimized in software. The UDP memory consistency model is simple; it "detects and stalls" conflicting references, ensuring that both complete, but in unspecified order. Thus, no complicated shared memory implementations are needed, and simple arbitration is used. Thus, the UDP enjoys fast local memory access and low access energy.

4. UDP System Interface

The UDP system API interfaces with the host programs. Note that UDP system API is different from the UDP lane ISA, which is used for constructing the UDP program for each lane. We will discuss the UDP lane ISA later. OP is passed by CPU general-purpose register.

write_udpreg

Opcode	Ops	Description
write_udpreg	OP1, OP2, OP3	write value to a single udp lane data register

OP1= laneid, OP2= regid
udplane[laneid].reg[regid] =OP3

read_udpreg

Opcode	Ops	Description
read_udpreg	OP1, OP2, OP3	read value from a single udp lane data register

OP1 stores "laneid", OP2 stores "regid"
OP3 = udplane[laneid].reg[regid]

pack_vec_udpreg

Opcode	Ops	Description
pack_vec_udpreg	OP1, OP2	pack value from a data register with same id of all udp lanes and stores it in a vector register

OP1 stores "regid", OP2 stores "vecid"

VRF[vecid]= (udplane[63].reg[regid], ……, udplane[0].reg[regid])

unpack_vec_udpreg

Opcode	Ops	Description
unpack_vec_udpreg	OP1, OP2	unpack value from a vector register in each data register of all UDP lanes

OP1 stores "regid", OP2 stores "vecid"
udplane[63].reg[regid] = VRF[vecid][2015-2047]
……
udplane[0].reg[regid] = VRF[vecid][0-31]

traverse

Opcode	Ops	Description
traverse	OP1, OP2, OP3	launch UDP execution

OP1 stores "start position" in the vector register, OP2 holds the length of the scan, OP3 holds the return value packed from each UDP lane's UDPReg. The UDP system scans vector register from position OP1 to position OP1 + OP2. The corresponding vector register for each lane is specified by the system configuration register.

launch

Opcode	Ops	Description
launch	OP1	start UDP execution

Each UDP lane starts at SBP from the value stored in local memory or vector register depending on vector-register/local-memory mode configuration. OP1 specifies the memory-mapped address reporting status of UDP system.

write_controlreg

Opcode	Ops	Description
write_controlreg	OP1	Set control registers of a single UDP lane. The value of each field in the control register is stored in memory pointed by OP1

read_controlreg

Opcode	Ops	Description
read_controlreg	OP1	Read control registers of a single UDP lane. The value of each field in the control register is stored in memory pointed by OP1.

write_activationQ

Opcode	Ops	Description
write_activationQ	OP1, OP2, OP3	Write an entry of the Activation Queue of a UDP lane

OP1 stores the “laneid”.

OP2 specifies the position in the Activation Queue.

OP3 stores the memory address that contain the value of the entry.

UDPlane[OP1].ActivationQueue.at(OP2) = *OP3

read_activationQ

Opcode	Ops	Description
read_activationQ	OP1, OP2, OP3	read an entry of the Activation Queue of a UDP lane

OP1 stores the “laneid”.

OP2 specifies the position in the Activation Queue.

OP3 stores the memory address that contain the value of the entry.

*OP3 = UDPlane[OP1].ActivationQueue.at(OP2)

write_system_config

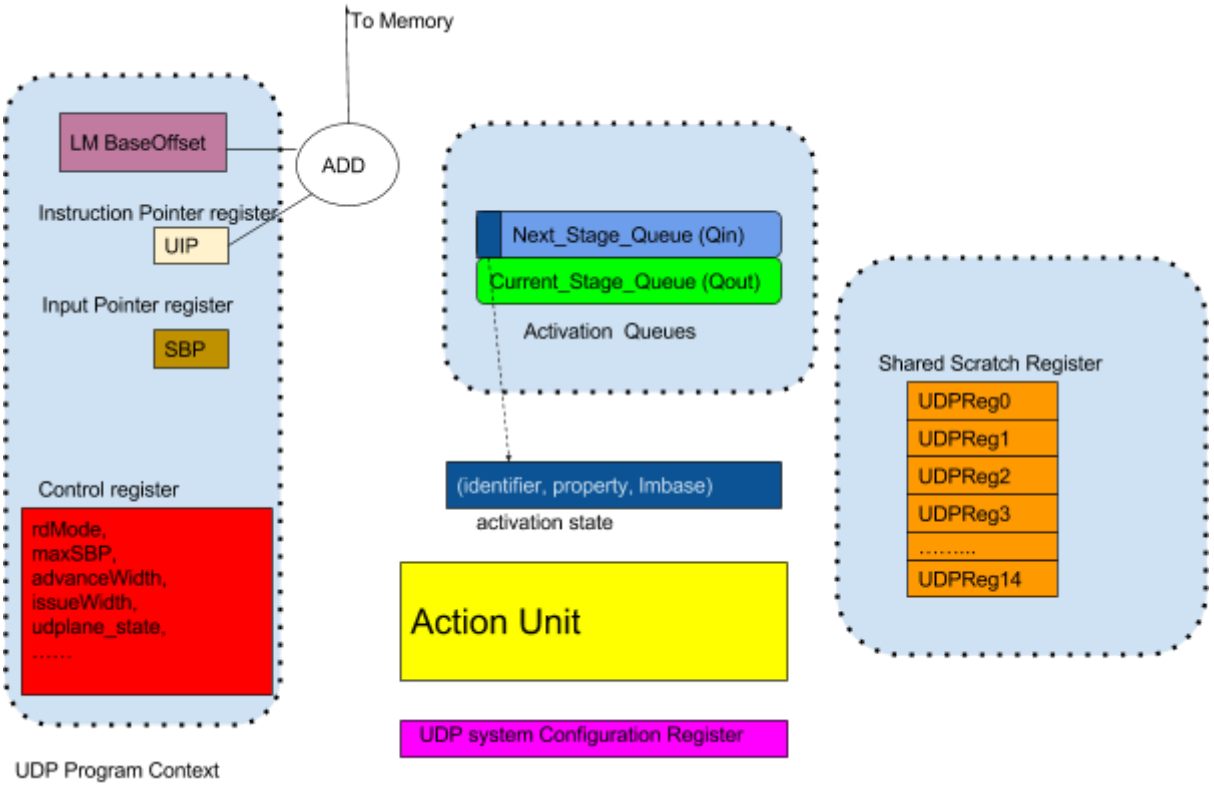
Opcode	Ops	Description
write_system_config	OP1	Configure the UDP system configuration register by the value in memory pointed by OP1

read_system_config

Opcode	Ops	Description
read_system_config	OP1	Read the UDP system configuration register and store it in memory pointed by OP1

5. UDP Lane Overview

This section explains the UDP lane architecture, instruction set, and implementations. UDP uses big-endian for local memory addressing.



5.1. Terms and Definitions

Activation: Activation is a UDP thread which is represented as (state identifier, property, lm base address).

Stage: All operations of current activations for current input word.

Stage Queue: In the Current Stage Queue (CSQ or Qout), each activation is related to the automaton state for the current SBP. In Next Stage Queue (NSQ, or Qin), each activation is related to the state for next SBP. The stage is associated the SBP. SBP points to the incoming symbol position in the vector register.

Transition: Automata transition. One of the two primitive categories in the UDP program. Functionality: Fast update the activation: 1) state identifier, and 2) state property.

Action: Operations associated with the transition primitive. Functionality: update UDP Lane state, including 1) UDP general purpose registers, 2) control registers, and 3) activations.

State Property: represented as (type, value) pair. "type" is the type of the state. "value" (if any) is the associated value. We explain the supported types and associated value below:

- *default*
  - If next stage signature check fails, transfer to another activation pointed by "value", and the same input word again. Repeat until signature success.
- *common*
  - No matter what input word issued, always fetch the same transition pointed by the state identifier.
- *majority*
  - if next stage signature check fails, construct the activation ("value", [majority-type, "value"]) as the next activation.
- *flag*
  - the state takes the value in UDPR0 as the label for transition in next stage. SBP doesn't advance.
- *persist*
  - the state is always on (active), never dies even next-stage signature fails.
  - In this way, no need to chain the self-to-self transition using epsilon transition to maintain state persistently.
- *NULL*
  - No type.
- *flag_majority*
  - state both has flag and majority type. "val" is the value of the majority transition address.

- *flag_default*
    - state both has flag and default type. “val” is the value of the default transition address.

Stream Buffer: holds the current 256-byte chunk of the input stream. It is a copy of the vector register specified by the Vector Register Mapping during configure time.

General Purpose Register: UDPReg[0] - UDPReg[14] holds general UDP 32-bit data value. UDPReg[15] is the alias of the SBP register.

Property Vector: Containing (type, value, lmBaseAddr) tuple for the state.

System Configuration Register (CfgR):
typedef struct{
vector_register_mapping: specify corresponding vector register to local stream buffer mapping (e.g. one-one mapping, one-to-all mapping);

active: whether the current UDP lane is active. If the lane is not active, it isn’t going to run.

} CfgR;

Control Register (CR):
typedef struct {
bit<16> UIP: UDP instruction-word pointer register that points to the word address of current executing UDP instruction (transition primitive or action primitive).

bit<4> lanes_fsm: Current internal state of the UDP lane.

bit<32> SBP: A 32-bit input stream pointer (bit address). In vector-register mode, SBP points to the stream buffer (a read-only copy of the vector register). In local-memory mode, SBP points to the local memory byte-address. In that

mode, the last 3 bits are always 0. In either mode, UDP lane offers the capability of:

- Completion detection:
  - Whenever SBP >= *maxSBP*
- Auto SBP increament
  - SBP automatically increase *CR.advance* when the stage finishes.

Note: we use the notation of SBPB to demonstrate the byte address. SBPB = SBP >> 3.

bit<32>maxSBP: whenever SBP >= *maxSBP*, UDP lane is marked as complete.

bit<4> issue_width: determines the next issued input word width. input word = buffer[SBP: SBP+*issue_width*]. Current supported *issue_width:*

- vector-register mode: 1,2,3,4,5,6,7,8;
- local-memory mode: 32

bit<3> advance_width: when each stage finishes, SBP increases "advance" steps (SBP += *advance_width*).

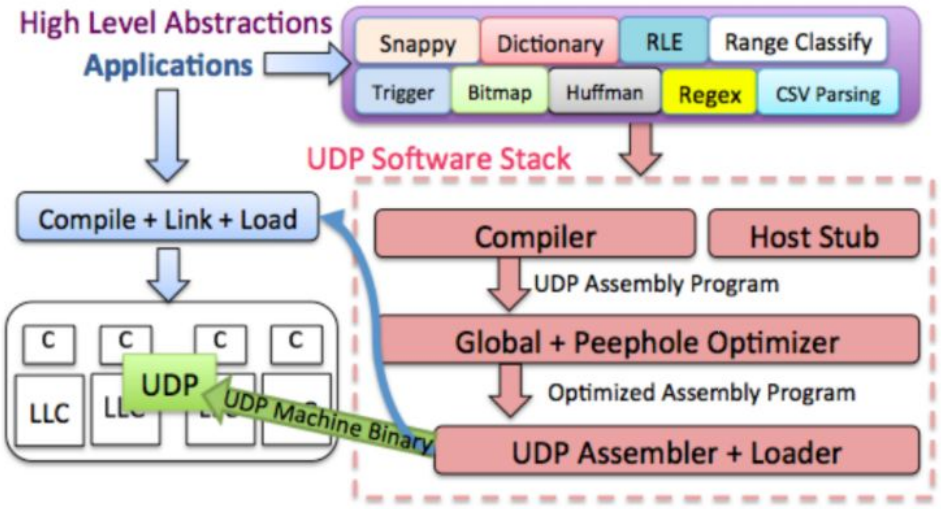
bit<1> LaneComplete: tells whether current UDP lane finishes.

bit<1> rdMode: 0 if vector-register mode, 1 if local-memory mode.

bit<1> stateDead: true if current activation is dead, false if it is alive.
} CR;

5.2. UDP Tool Chain

This section shows the software architecture of the UDP tool chain for translating a high-level computation abstraction into the UDP-lane program. A 64-lane UDP system executes 64 UDP programs in parallel.



A number of domain-specific translators and a shared backend are used to create the UDP programs used for application kernels. The translators support a high-level abstraction, and translate it into a high-level assembly language. The backend does intra-block and cross-block optimization, but most importantly, it does the layout optimization to achieve high code density with multi-way dispatch. Further, it optimizes action block sharing, another critical capability for small code size. Finally, the system stubs for linking with CPU programs, enabling flexible combination of CPU and UDP computing.

5.3. UDP Lane Execution Sequence

Each UDP lane runs asynchronously after during each *launch()* call. Each activation starts with the transition primitive. Each activation must at least contain 1 transition.
Activation starts: [Transition1] → [action1.1] → [Transition2] → [action2.1] → [action2.2] → .....→ [actionx.y] Activation ends.

6. UDP Lane ISA

6.1. UDP Assembly Grammar

See the "UDP Assembly" section for detail assembly-level transition primitives and actions description. In this section, we describe how these primitives are written for program construction.

S → E S | ε

E → Hybrid_Tran | Shared_Block

Shared_Block → BLOCK_NUM {Action_List}

Hybrid_Tran → T;Action_List

Action_List → A; Action_List | ε

T → labeled_TX | default_TX | majority_TX | epsilon_TX | flagged_TX | common_TX

A → ImmAction | Imm2Action | RegAction

labeled_TX → LABELED_TX (src, label, dst)

default_TX → DEFAULT_TX (src, def_state)

majority_TX → MAJORTY_TX (src, majority_state)

epsilon_TX → EPSILON_TX (src, dst)

flagged_TX → FLAGGED_TX (src, flag, dst)

common_TX → COMMON_TX (src, dst)

ImmAction → opcode src, dst, imm

Imm2Action → opcode src, dst, imm, imm2

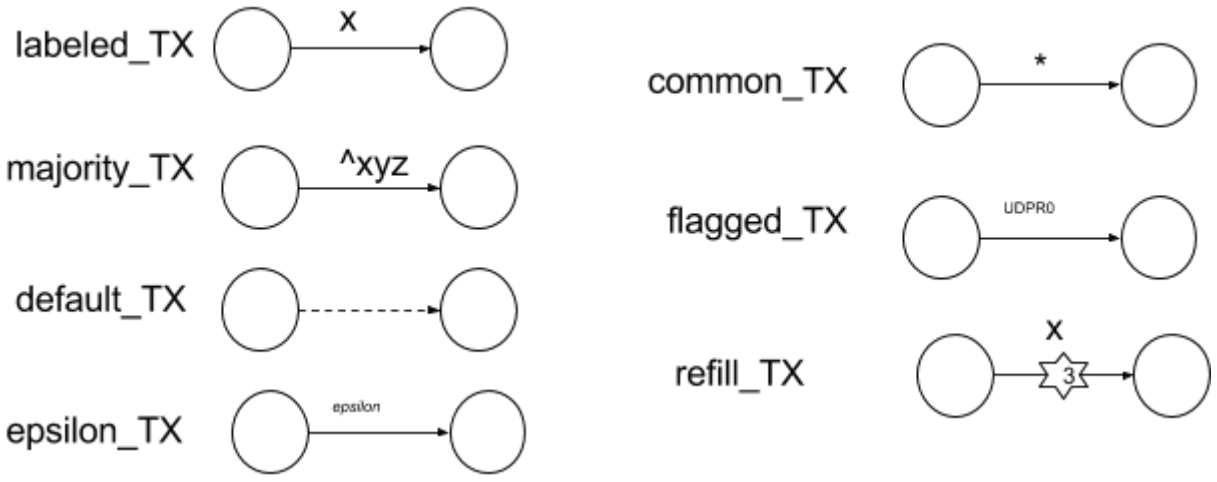
RegAction → opcode src, ref, dst

6.2. UDP Assembly Instruction

6.2.1. Transition Primitive

There are total 7 transition primitive types in UDP. Each of them address different goals: 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.



Labeled_TX	[current_active_state-----input_word------>next_active_state]
Default_TX	[current_active_state------>default_state]
Flagged_TX	[current_active_state-----UDPR0------>next_active_state]
Epsilon_TX	[next_active_state_1------>next_active_state_2]
Majority_TX	[current_active_state---- NOT IN [input_word_1, input_word_2 ,..., input_word_X] ---->next_active_state]
Common_TX	[current_active_state------->next_active_state]
Refill_TX	[current_active_state-----input_word------>next_active_state] SBP rollback x bits. x = 1-8, which is specified in the transition

6.2.2. Action

set_state_property $property	set next_active_state’s property ($type, $value)
fork_state $state_ident, $property	create a new next_activation = ($state_ident, $property, lane.lmBase)
set_issue_width $width	CR.issue_width = $width

put_2bytes_imm UDPR, $bytes	LM[UDPR].write2Byte($bytes); UDPR+=2
put_1byte_imm UDPR, $bytes	LM[UDPR] = write1Byte($bytes); UDPR+=1
put_bytes UDPRs, UDPRd, $len	X = $len; LM[UDPRd] = writeXByte($bytes); UDPR+=X;
get_bytes UDPRs, UDPRd, $len	X = $len; UDPRd = LM[UDPRs].readXByte(); UDPRs+=X;
put_bits UDPR,$bits,$len	X = $len; LM[UDPR].writeXBit($bits); UDPR+=X
get_bits UDPRs, UDPRd ,$len	X = $len; UDPRs = LM[UDPRs].readXBit(); UDPRs+=X
compare_string Rs, Rt, UDPRd	UDPRd←compare_str(Rs, Rt). No update on Rs, Rt.
copy UDPRs, UDPRt, UDPRd	copy(UDPRs, UDPRd, UDPRt). update UDPRs, UDPRt and UDPRd. UDPRd=0 if copy finish
copy_imm UDPRs, UDPRd, $length	copy(UDPRs, UDPRd, $length). update UDPRs and UDPRd.
lshift_or UDPRs, UDPRd , $shift	UDPRd =UDPRd | (UDPRs << $shift)
rshift_or UDPRs, UDPRd , $shift	UDPRd =UDPRd | (UDPRs >> $shift)
lshift_and UDPRs, UDPRd , $shift	UDPRd =UDPRd & (UDPRs << $shift)
rshift_and UDPRs, UDPRd , $shift	UDPRd =UDPRd & (UDPRs >> $shift)
lshift_or_imm UDPRs, UDPRd , $shift, $imm	UDPRd =$imm | (UDPRs << $shift)
rshift_or_imm UDPRs, UDPRd , $shift, $imm	UDPRd =$imm | (UDPRs >> $shift)
lshift_and_imm UDPRs, UDPRd , $shift, $imm	UDPRd =$imm & (UDPRs << $shift)
rshift_and_imm UDPRs, UDPRd , $shift, $imm	UDPRd =$imm & (UDPRs >> $shift)
lshift_add_imm UDPRs,UDPRd,$shift, $imm	UDPRd =$imm + (UDPRs << $shift)
lshift_sub_imm UDPRs,UDPRd,$shift, $imm	UDPRd = (UDPRs << $shift) - $imm
rshift_add_imm UDPRs,UDPRd,$shift, $imm	UDPRd =$imm + (UDPRs >> $shift)
rshift_sub_imm UDPRs,UDPRd,$shift, $imm	UDPRd = (UDPRs >> $shift) - $imm
hashsb32 UDPRd , $HTBASE	UDPRd = hash(sb[SBP]) + $HTBASE or UDPRd = hash(LM[SBP]) + $HTBASE
addi Rs , Rd, $imm	Rd← Rs + $imm

subi Rs , Rd, $imm	Rd← Rs - $imm
add Rs , Rt, Rd	Rd ← Rs + Rt
sub Rs , Rt, Rd	Rd ← Rs - Rt
mov_lm2reg UDPRs , UDPRd, $bytes	UDPRd ← LM[DS+UDPRs : DS+UDPRs+3] & mask $bytes=1, mask = 0xff $bytes = 2, mask = 0xffff $bytes =3, mask = 0xffffff $bytes =4, mask = 0xffffffff
mov_reg2lm UDPRs , UDPRd,$bytes	LM[DS+UDPRd : DS+UDPRd +$bytes-1] ← UDPRs & mask
mov_sb2reg UDPRd	UDPRd ← stream_buffer[SBP:SBP+CR.issue]
mov_reg2reg UDPRs, UDPRd	UDPRd ← UDPRs
mov_imm2reg UDPRd, $imm	UDPRd ← $imm
comp_lt UDPRs, UDPRd, $imm	UDPRd ← UDPRs < $imm?
comp_gt UDPRs, UDPRd, $imm	UDPRd ← UDPRs > $imm?
comp_eq UDPRs, UDPRd, $imm	UDPRd ← UDPRs = $imm?
compreg_lt UDPRs, UDPRt, UDPRd	UDPRd ← UDPRs < UDPRt?
compreg_gt UDPRs, UDPRt, UDPRd	UDPRd ← UDPRs > UDPRt?
compreg_eq UDPRs, UDPRt, UDPRd	UDPRd ← UDPRs = UDPRt?
TranCarry_goto $BLOCK_ID	UIP ← address of symbolic shared action block $BLOCK_ID. In Machine Code, it is implemented as a transition pritmive
goto $BLOCK_ID	UIP ← address of symbolic shared action block $BLOCK_ID.
refill $imm	When stage finish, SBPB ← SBPB - $imm + CR.Advcance
bitwise_and_imm UDPRs, UDPRd, $imm	UDPRd← UDPRs & $imm
bitwise_and UDPRs, UDPRt, UDPRd	UDPRd← UDPRt & UDPRs
bitwise_or_imm UDPRs, UDPRd, $imm	UDPRd← UDPRs | $imm
bitwise_or UDPRs, UDPRt, UDPRd	UDPRd← UDPRs | UDPRt

6.3. UDP Machine Instruction

The UDP program's IR is in the form of UDP assembly instructions. It is also known as extended finite automata form (EFA). However, in the hardware implementation, all assembly-level transition information must be associated with the state. The mapping between UDP assembly instruction and UDP machine instruction fulfills this job. We describe the machine instruction and their relationship with assembly-level instruction below.

6.3.1. Transition Primitive

basic_TX	dst_state property is basic. "value" not carried. It can be labeled or non-labeled transition
basic_with_action_TX	dst_state property is basic. Transition associated with actions. "value" not carried
Epsilon_TX	fork another activation for next stage.
majority_carry_TX	dst_state property is majority. "value" carried
default_carry_TX	dst_state property is default. "value" carried
persistent_carry_TX	dst_state property is persistent. "value" not carried
Refill_TX	dst_state property is basic. "value" not carried. But update SBP to allow partial prefix-free symbols
Refill_with_action_TX	dst_state property is basic. "value" not carried. But update SBP to allow partial prefix-free symbols. Transition associated with actions
flag_carry_TX	dst_state property is flag. "value" not carried
flag_carry_with_action_TX	dst_state property is flag. "value" not carried. Transition associated with actions
common_carry_TX	dst_state property is common. "value" not carried
common_carry_with_action_TX	dst_state property is common. "value" not carried. Transition associated with actions
flagmajority_carry_TX	dst_state property is flagmajority. "value" is majority transition address
flagdefault_carry_TX	dst_state property is flagdefault. "value" is default

	transition address

6.3.2. Action

Same as Logical Actions in "UDP Assembly Instruction" section, except for TranCarry Actions:

- TranCarry_goto $BLOCK_ID is implemented as basic_with_action_TX. Address of the block.$BLOCK_ID is specified in attach field
- TranCarry_refill $imm is implemented as refill_TX or refill_with_action_TX. #imm is specified in part of attach field

The assembler translates assembly instruction into machine instruction. The following Table shows the relationship. *Signature* is to verify the fetched transition is indeed the desired one.

Assembly Instruction	Machine Instruction
Label_TX	basic_TX with current issue word signature success
Default_TX	1. default_carry_TX for previous incoming transition 2. signature fail for current issue word 3. basic_TX pointed by "value" of current state property, do signature check. If fail, re-do step 1-3. If success, transition ends
Flagged_TX	1.flag_carry_TX for previous incoming transition 2.Basic_TX with UDPR0 as label. Signature must be success
Epsilon_TX	Basic_TX pointed by physical Epsilon_TX. No Signature check
Majority_TX	1. majority_carry_TX for previous incoming transition 2. signature fail for current issue word 3. Basic_TX pointed by "value" of current state property. No Signature check
Common_TX	1. common_carry_TX for previous incoming transition. 2. Basic_TX pointed by current state identifier.

	No Signature Check
Refill_TX	1. Refill_TX with current issue word signature success and refill value packed in the transition
Refill_TX[with_action]	Refill_TX with current issue word signature success and refill value packed in the transition and action address packed
Label_TX[with_action]	basic_with_action_TX or XXXcarry_with_action_TX with current issue word signature success
Default_TX[with_action]	1. default_carry_TX for previous incoming transition 2. signature fail for current issue word 3. basic_with_action_TX pointed by “value” of current state property. do signature check. If fail, re-do step 1-3. If success, start to do actions.
Flagged_TX[with_action]	1.flag_carry_TX or flag_carry_with_action_TX for previous incoming transition 2. Basic_with_action_TX with UDPR0 as label. Signature must be success
Epsilon_TX[with_action]	Basic_with_action_TX pointed by physical Epsilon_TX. No Signature check
Majority_TX[with_action]	1. majority_carry_TX for previous incoming transition 2. signature fail for current issue word 3. Basic_with_action_TX pointed by “value” of current state property. No Signature Check
Common_TX[with_action]	1. common_carry_TX or common_carry_with_actions_TX for previous incoming transition. 2. Basic_with_action_TX pointed by current state identifier. No Signature Check

7. Physical Implementation

7.1. Transition Encoding

signature(8)	target(12)	type(4)	attach(8)

typedef struct {
signature(8): verify the correctness of physical transition primitive word;
target(12): state identifier;
type(4): UDP transition primitive type;
attach(8): auxiliary information place holder
} transition_primitive;

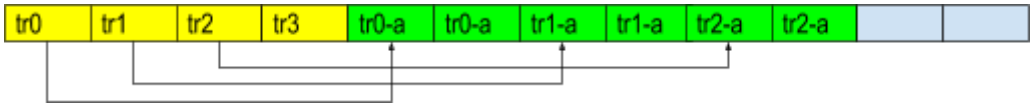
We list the transition type and the corresponding attach usage:

type (4)	Symbolic	attach (8)
0	BASIC	unused
1	EPSILON	Epsilon transition address
2	REFILL	refill (3b)
3	MAJORITYCARRY	majority transition address
4	DEFAULTCARRY	default transition address
5	FLAGCARRY	unused
6	COMMONCARRY	unused
7	PERSISTCARRY	unused
8	FLAGMAJORITYCARRY	majority transition address
9	FLAGDEFAULTCARRY	default transition address
10	BASIC_WITH_ACTION	mode(2b) base(3b) scalar (3b)
11	REFILL_WITH_ACTION	refill (3b) base(3b) scalar (2b)
12	FLAG_WITH_ACTION	mode(2b) base(3b) scalar (3b)
13	COMMON_WITH_ACTION	mode(2b) base(3b) scalar (3b)
14		
15		

7.1.1. Addressing Mode

- mode = 11.

- Design for a state whose outgoing transitions all have roughly same actions



- relative_addr = BASE[base_idx] + signature (8b) << SCALAR[scalar_idx]
  - BASE = {1,2,4,8,16,64,256} for base_idx=0-6
  - base_idx = 7, relative_addr = 1
  - 2^SCALAR[scalar_idx] of actions per tran . SCALAR = {0,1,2,3,4,5,6,7}
- absolute address = LM_BASE + UIP + relative_addr

- mode = 00, 01, 10
  - whole “attach” field serves as the offest points to the current LM bank
  - Addressing range: [LM_BASE, LM_BASE + 191]

For REFILL_WITH_ACTION, mode = 11, scalar is LSB 2 bits and assume MSB = 0

7.2. Action Encoding

Three types of action: ImmAction, Imm2Action, RegAction.

7.2.1. ImmAction

opcode(7)	last(1)	src(4)	dst(4)	imm(16)

typedef struct {
opcode(7): action opcode
src(4): source UDP register ID
dst(4): dstination UDP register ID
imm(16): immediate number
last(1): action list end
} ImmAction;

7.2.2. Imm2Action

opcode(7)	last(1)	src(4)	dst(4)	imm(4)	imm2(12)

typedef struct {
opcode(7): action opcode
src(4): source UDP register ID
dst(4): dstination UDP register ID
imm2(12): immediate number
imm(4): short immediate number
last(1): action list end
} Imm2ction;

7.2.3. RegAction

opcode(7)	last(1)	src(4)	ref(4)	dst(4)	

typedef struct {
opcode(7): action opcode
src(4): source UDP register ID
dst(4): dstination UDP register ID
ref(4): reference UDP register ID
last(1): action list end
unused(12): unused bits
} RegAction;

7.3. Hardware Execution Sequence

Start_Current_Stage:
while CR.CSQ_empty == FALSE
    activation = CSQ.pop();
    Update_UIP (activation, stream_buffer[SBP:SBP+CR.issue];
    UDP_INST = LM[DS+UIP];
    Execute (UDP_INST);

Inc_SBP(SBP,CR, UDP_INST);
CSQ = NSQ;
clear NSQ;
Goto *State_Current_Stage*

7.3.1. Update_UIP Description

1. For basic/refill/persistent property state:
   Update_UIP (*activation*, InputWord):
   {UIP = $target + InputWord;}
2. For common property state:
   Update_UIP (*activation*, InputWord):
   {UIP = $target;}
3. For flag/flagmajority/flagdefault property state:
   Update_UIP (*activation*, InputWord):
   {UIP = $target + UDPR0;}

7.3.2. Inc_SBP Description

SBP = SBP + CR.issue_width - UDP_INST.$refill;

7.3.3. Execute Description

We describe the detail of Execute (UDP_INST) for 7 UDP assembly-level transition types in Section "Examples on Assembly Instruction on Hardware".

7.4. Machine-level Transition Operation

This section describes how does machine-code transition change UDP lane architectural state. Note that *fmode()* means use the addressing scheme in Section "Hardware Implementation".

Signature check is to check whether "input_word" == fetched_tran.signature.

- If signature_check == true (the fetched transition is the one we need):

basic_TX	tran = LM[cur_activation.stateID + "input"] activation = (tran.target, [P_NULL, NULL])

basic_with_action_TX	tran = LM[cur_activation.stateID + “input”] activation = (tran.target, [P_NULL, NULL]) first action = LM [f*mode*(tran.attach)]
Epsilon_TX	tran = LM [tran.attach]
majority_carry_TX	tran = LM[cur_activation.stateID + “input”] activation = (tran.target, [majority_type, tran.attach])
default_carry_TX	tran = LM[cur_activation.stateID + “input”] activation = (tran.target, [default_type, tran.attach])
persistent_carry_TX	tran = LM[cur_activation.stateID + “input”] activation = (tran.target, [persistent_type, NULL])
flag_carry_TX	tran = LM[cur_activation.stateID + “input”] activation = (tran.target, [flag_type, NULL])
flag_carry_with_action_TX	tran = LM[cur_activation.stateID + “input”] activation = (tran.target, [flag_type, NULL]) first action = LM [f*mode*(tran.attach)]
common_carry_TX	tran = LM[cur_activation.stateID + “input”] activation = (tran.target, [common_type, NULL]
common_carry_with_action_TX	tran = LM[cur_activation.stateID + “input”] activation = (tran.target, [common_type, NULL]) first action = LM [f*mode*(tran.attach)]
flag_majority_TX	tran = LM[cur_activation.stateID + “input”] activation = (tran.target, [flag_majority_type, majority_tran_addr])
flag_default_TX	tran = LM[cur_activation.stateID + “input”] activation = (tran.target, [flag_defualt_type, default_tran_addr])
refill_TX	tran = LM[cur_activation.stateID + “input”] activation = (tran.target, [P_NULL, NULL]) SBP = SBP - tran.attach.refill_val
refill_with_action_TX	tran = LM[cur_activation.stateID + “input”] activation = (tran.target, [P_NULL, NULL]) first action = LM [f*mode*(tran.attach)]

- If signature_check == false:

- If current activation.property.type == “majority_type (P_MAJORITY)” or “flagmajority_type (P_FLAGMAJORITY)”, fetch majority transition, the address is activation.property.value
- If current activation.property.type == “default_type (P_DEFAULT)” or “flagdefault_type (P_FLAGDFEFAULT)”, fetch default transition, the address is activation.property.value
- Otherwise, mark the activation is dead, set CR.stateDead = true.

7.5. Machine-level Action Operation

*** It is one-one Mapping for hardware implementation actions

7.6. Assembly-level Transition on Hardware

This section describes how assembly transition primitives are implemented by physical transition (machine instruction) and mapped to hardware; Each implementation has a figure that shows the idea. The green lines show the logical transition (assembly instruction). The purpose of this section is to make the relationship of UDP assembly(logical) transition, UDP machine(physical) transition, and hardware implementation more concrete.

We first define a function here for convenience.

```
SetProperty(UDP_INST):
        p = NewProperty()
        p.type = null;
        if UDP_INST.type == DEFAULT_IDENT/DEFAULT_ADDR:
                p.default = UDP_INST.attach;
                p.type = UDP_INST.type;
        if UDP_INST.type == MAJ_IDENT/MAJ_ADDR:
                p.majority = UDP_INST.attach;
                p.type = UDP_INST.type;
        if UDP_INST.type == COMMON:
                p.type = UDP_INST.type;
                CR.hold = TRUE;
        if UDP_INST.type == FLAGED:
                p.type = UDP_INST.type;
```

return p;

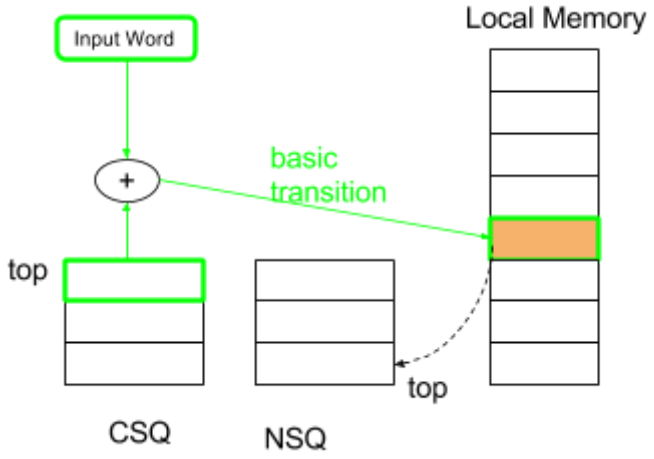
1. Label transition (Label_TX):

UDP Instruction (UDP_INST):

** Don't care **

Description:

Deterministic FA transition. The signature has to succeed in order to trigger a basic transition. It is implemented as a physical basic_TX with a signature check success.



Execute (UDP_INST):

```
{
    assert(CR.signature_check==TRUE);
    p = SetProperty(UDP_INST);
    NSQ.push([$target, p]);
}
```

2. Default transition (Default_TX):

UDP Instruction (UDP_INST):

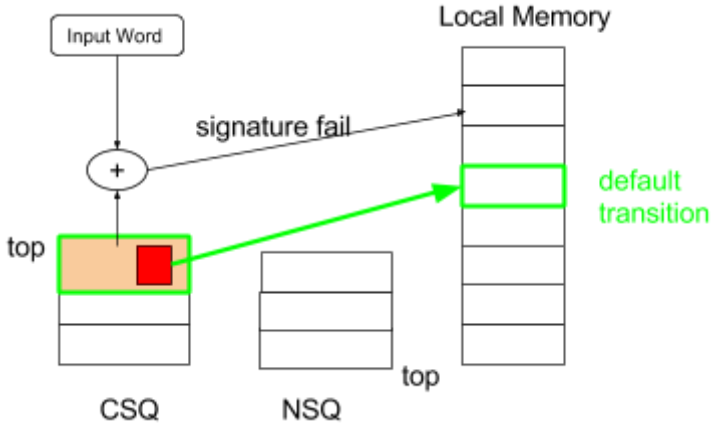
*****Don't care*****

Activation:

state identifier	property			
$target	…….	$default	$type: DEFAULT	…….

Description:

Ignore the current fetched UDP_INST from LM due to signature failure. Logical Default_TX is implemented as first use default_carry_TX or default_carry_with_action or an action to acquire the default property and triggers the default_TX through signature fail by a basic_TX in next stage. Optimization happens in *activation*.property.$type: if it is DEFAULT_IDENT, $default serves the state identifier of the default state. The default transition depth is limited as 1 by this optimization. if it is DEFAULT_ADDR, $default serves as the address of the default transition. No action followed. The figure plots the situation when $type = DEFAULT_ADDR.



Execute (UDP_INST):

```
{
    assert (CR.signature_check==FALSE,
    activation.propert.type = DEFAULT_ADDR or DEFAULT_IDENT);
    IF activation.property.type == DEFAULT_ADDR:
        UIP = activation.property.default;
        UDP_INST = LM[CS + UIP];
        UIP = UDP_INST.target +
            stream_buf [SBP:SBP+CR.issue_width];
        UDP_INST = LM[CS + UIP ];
    ELSE:
        UIP = activation.property.default +
            stream_buf [SBP:SBP+CR.issue_width];
        UDP_INST = LM[CS + UIP ];
    Execute (UDP_INST);
```

}

3. Flagged transition (Flagged_TX):

UDP Instruction (UDP_INST):

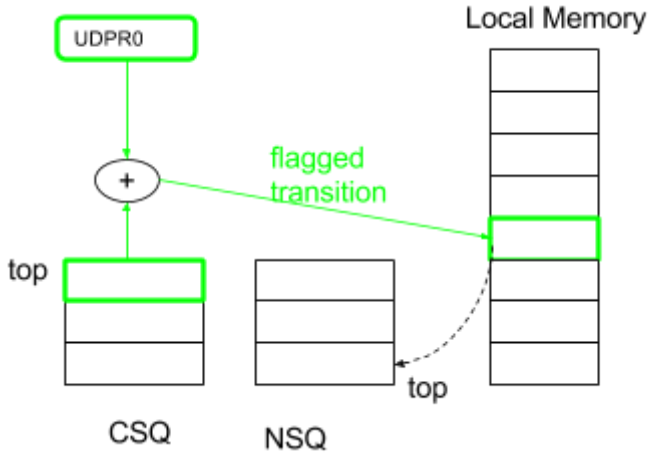
**** Don't Care ****

Activation:

state identifier	property			
$target	…….	……..	$type: FLAGED	…….

Description:

Flagged transition is implemented as first use flag_carry_TX, flag_carry_with_action or action to acquire the flag property. Then using a basic_TX and take UDPR0 as the label.



Execute (UDP_INST):

{

p = SetProperty(UDP_INST);

NSQ.push([$target, p]);

}

4. Epsilon transition:

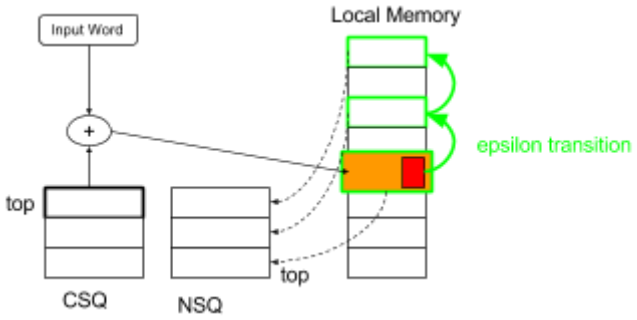
UDP Instruction (UDP_INST):

signature	target	type	refill	attach
	$target	EPSILON_ADDR/	$refill	$attach

		EPSILON_IDENT		

Description:

Epsilon transition. Logical Epsilon transitions are implemented by physical Epsilon transition. Physical Epsilon transition is chained through $attach, which serves as a pointer (EPSILON_ADDR) to the physical epsilon transition (can have other epsilon transition chained) or state identifier (EPSILON_IDENT) for the only one epsilon transition (no further epsilon transition is allowed). The figure only plots the EPSILON_ADDR.



Execute (UDP_INST):

```
{
    assert( CR.signature_check==TRUE);
    NSQ.push(UDP_INST);
    IF $type == EPSILON_ADDR:
        UIP = $attach;
        UDP_INST = LM[CS + UIP];
        Execute (UDP_INST);
    ELSE IF $type == EPSILON_IDENT:
        UDP_INST=(na, $attach, BASIC, $refill, na);
        p = SetProperty(UDP_INST);
        NSQ.push([$target, p]);
}
```

5. Majority transition (Majority_TX):

UDP Instruction (UDP_INST):

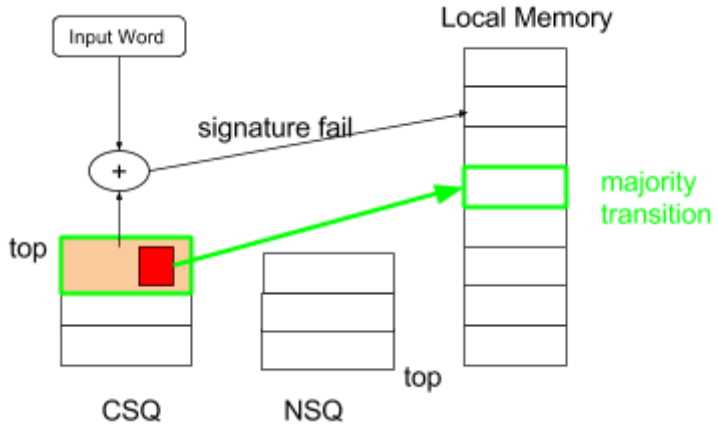
*****Don't care*****

Activation:

state identifier	property			
	…….	$majority	$type:MAJ_IDENT/MAJ_ADDR	…….

Description:

Ignore the current fetched UDP_INST from LM due to signature failure. Majority transition is implemented as first use majority_carry_TX or majority_carry_with_action_TX or action to acquire the majority property. Then a signature fails on the next stage triggers the majority transition. $majority serves as a pointer (MAJ_ADDR) to the majority transition or state identifier (MAJ_IDENT) for the majority transition whose majority transition is itself.



Execute (UDP_INST):

```
{
    assert(CR.signature_check==FALSE);
    IF activation.property.type == MAJ_ADDR:
        UIP = activation.property.majority;
        UDP_INST = LM[CS + UIP];
    ELSE:
        UDP_INST = (na, activation.property.majority, MAJ_IDENT,
                    $refill, activation.property.majority);
    p = SetProperty(UDP_INST);
    NSQ.push([$target, p]);
}
```

6. Common transition:

UDP Instruction (UDP_INST):

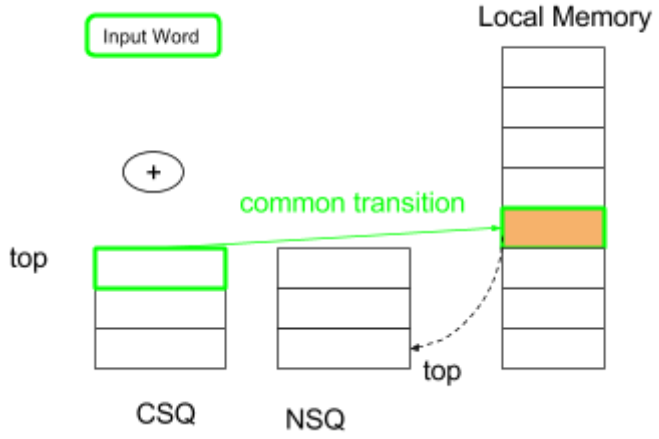
**** Don't Care ****

Activation:

state identifier	property			
$target	…….	……..	$type: Common	…….

Description:

The logical Common transition is implemented as first use common_carry_TX or common_carry_with_action or action to acquire the "common" state property. Then in the next stage, whatever the input word is, the transition is always the same (pointed by the current state identifier).



Execute (UDP_INST):

```
{
        p = SetProperty(UDP_INST);
        NSQ.push([$target, p]);
}
```

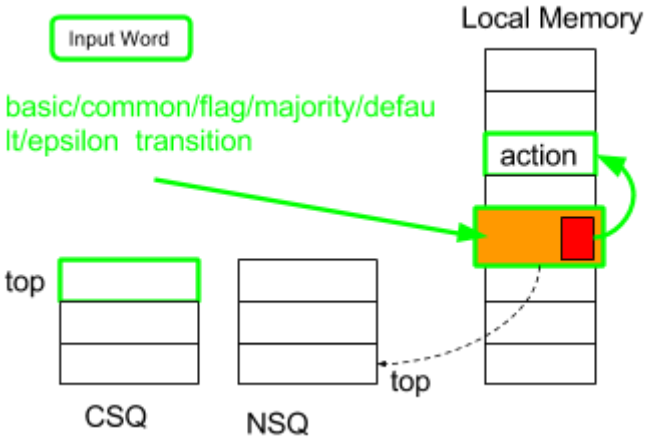
7. Basic/Common/Flagged/Majority/Default/Epsilon Transition with Actions:

UDP Instruction (UDP_INST):

signature	target	type	refill	attach
	$target	ACTIONS	$refill	$attach

Description:

The first action of the action list is pointed by the $attach. For the "value" need to be carried, use an action since the $attach field is occupied as a pointer to actions.



Execute (UDP_INST):

```
{
        assert(CR.signature_check==TRUE);
        p = SetProperty(UDP_INST);
        NSQ.push([$target, p]);
        UIP = UIP + $attach;
        UDP_INST = LM[CS + UIP];
        CR.word_type == ACTION;
        DoActions(UDP_INST);
}
```

7.6.1. DoActions Description

Start_Action

```
IF UDP_INST.last == TRUE:
        CR.word_type == PRIMITIVE;
ELSE:
        ExecuteAction(UDP_INST);
        UIP = UIP + 1;
        UDP_INST = LM[CS + UIP];
        Goto Start_Action;
```

7.6.2. ExecuteAction Description

Actions not discussed here are straight-forward implementations. Referring the "UDP Assembly Instruction" section for explanation.

1. Comparison

   Compare UDPR3, UDPR2, UDPR1

   - Description:

     It compares data starting from byte address UDPR3 with data starting from byte address UDPR2. The length of the same prefix part is returned in UDPR1.

   - Operation:

     inout: UDPR3, byte address

     inout: UDPR2, byte address, points to last byte that matches reference stream.

     output: UDPR1. Length of the match in bytes

     {

     UDPR1 = 0; ref_address = UDPR3; src_address = UDPR2;

     WHILE LM[ref_address++] == LM[src_address++]

     ++UDPR1;

     return UDPR1, UDPR3=ref_address, UDPR2=src_address;

     }

2. Copy

   Copy UDPR1, UDPR2, UDPR3

   - Description:

     It copies data starting from byte address UDPR1 to the byte address UDPR2. The end of the source stream is specified in UDPR3.

   - Operation:

     inout: UDPR1. Source byte address.

     inout: UDPR2. Destination byte address.

     input: UDPR3. End source byte address.

     {

     src_address = UDPR1; dst_address = UDPR2; end_address = UDPR3;

```
        while (src_address != end_address)
                LM[dst_address++] = LM[src_address++]
        UDPR1 = src_address; UDPR2 = dst_address;
        return UDPR1, UDPR2;
}
```

3. Set Issue Width

   SetIssue $width

   - Description:

     Set the CR.issue_width to $width

   - Operation:

     input: $width. Action stored in LM and $width is a subfield of action

{CR.issue_width = $width; return;}

4. Put Bytes

   PutBytes UDPR1, $symbol, $len,

   - Description

     It puts the $symbol in the byte address UDPR1 in LM.

     input: $symbol

     input: $len. size of symbol in terms of byte

     inout: UDPR1. dstination byte address

   - Operation

```
{
        LM[UDPR1:UDPR1+$len] = $symbol;
        UDPR1 +=  $len;
        return UDPR1;
}
```

5. Put Bits

   PutBit UDPR1, $bits, $len

   - Description

     It puts a sequence of bits in LM starting from UDPR1 and updates UDPR1. The length of the bits is specified in $len.

   - Operation

     input: $bits

```
        input: $len. size of bits in terms of bit
        inout: UDPR1.

{
        LM[UDPR1:UDPR1 + $len] = $bits;
        UDPR1 += $len;
        return UDPR1;
}
```

8. Summary

This document provides detail description about UDP system architecture organization, host system integration interfaces, UDP lane architecture and ISA, and microarchitecture implementation for transition primitives and action primitives. It also serves a detail explanation of the academic literature [1].

9. Reference

[1]. Yuanwei Fang, Chen Zou, Aaron J. Elmore, and Andrew A. Chien. "UDP: A Programmable Accelerator for Extract-Transform-Load Workloads and More" in Proc. 50th Annual IEEE/ACM International Symposium on Microarchitecture, October 2017.

[2]. Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. "Fast support for unstructured data processing: the unified automata processor" in Proc. 48th Annual IEEE/ACM International Symposium on Microarchitecture, December 2015.

[3]. Yuanwei Fang, Andrew Lehane, and Andrew A. Chien, "EffCLiP: Efficient coupled-linear packing for finite automata", University of Chicago Technical Report, TR-2015-05, May 2015.