Before jumping into the use case some processing steps have to be analyzed in terms of CPU and memory requirements. After this is done the use case can be put together using these processing steps.

5.1 Processing steps

In this section we use the ARM Cortex-M3 as a reference design [15], but any RISC core with division support does the job roughly the same way. We use assembly code fragments where necessary to clearly show the cycle count and memory use of the processing steps.

1. *Packet copy between I/O and memory:*

In a direct L1 memory transfer the L1 SRAM can store 4 bytes (on 32 bit systems) per clock cycle. Usually only the header is copied to L1 memory, with a maximal size of 64 bytes, meaning 16 clock cycles.

With L2 or external DRAM 1-24 accesses are needed based on packet length (according to the model as described at the "model limitations" section). Orchestrating memory areas is not included here. We assume simple (hardware) logic for selecting the right core and/or the right queue.

So packet send/receive needs 16 core clock cycles plus 12 DRAM accesses.

2. *Parsing a register-wide data field from the header (in L1 memory) on fixed offset:*

```
Register $1: header pointer
Register $2: result
Code:
       ldr $2, [$1, <offset>]
```

So one assembly instruction is required and usually 2 core clock cycles.

3. *Direct lookup from a byte array:*

```
Register $1: array pointer
Register $2: index
Register $3: result
Code:
       add $4, $1, $2
       ldrb $3, [$4]
```

So one add and one load instruction is needed plus the additional memory operation. This is 3 core clock cycles plus 1 DRAM access.

4. *Simple hash table lookup:*

Note that this calculation is valid for a simple XOR based hash key and direct hash.

```
Register $1-N: key components
Register $HL: hash length
Register $HP: hash array pointer
Register $HE: hash entry size
Register $Z: result
Pseudo code:
       # hash key calculation
       eor $tmp, $tmp
       for i in 1 … N
              eor $tmp, $i
       # key is available in $tmp

       # calculate hash index from key
       udiv $tmp2, $tmp, $HL
       mls $tmp2, $tmp2, $HL, $tmp
       # index is available in $tmp2

       # index -> hash entry pointer
       mul $tmp, $tmp2, $HE
       add $tmp, $HP
       # entry pointer available in $tmp

       <prefetch entry to L1 memory>
       # pointer to L1 entry -> $tmp2

       # hash key check (entry vs. key)
       for i in 1 … N
              ldr $Z, [$tmp2], #4
              # check keys
              cmp $i, $Z
              bne collision
       # no jump means matching keys
       # pointer to data available in $Z
```

As it is visible, the resource requirement of a hash lookup depends on the complexity of the key. A good approximation could be:

$$Cycles = N * 5 + 14 + \frac{HE}{4} + 1 * mem.access$$

Note that this does not contain collision resolution. That would roughly double the effort, but we assume the probability is low enough.

Some typical examples:

- key with 2 components: {VLAN, DMAC}
- key with 5 components: TCP/IP 5 tuple {saddr, daddr, proto, sport, sport}

5.2 Evaluating the PBB use case

Using the building block analysis above we can now construct the PBB use case and by using the model parameters evaluate the performance of our artificial NPU.

For a PBB switch roughly the following steps are required:

1. Read Ethernet frame from I/O, store header in L1, payload in L2 memory *(~16 cycles + 12 L2 memory write in average)*
    - Note that for small packets (e.g. TCP ACK) we can entirely skip using L2 memory
2. Parse fixed parameters from header *(6 cycles)*
    - physical ingress port
    - VLAN ID
    - DMAC
3. Find extended VLAN {source physical port, VLAN ID → eVLAN}: hash lookup *(~26 cycles + 1 L2 memory access)*
    - 2 key components (~uint32)