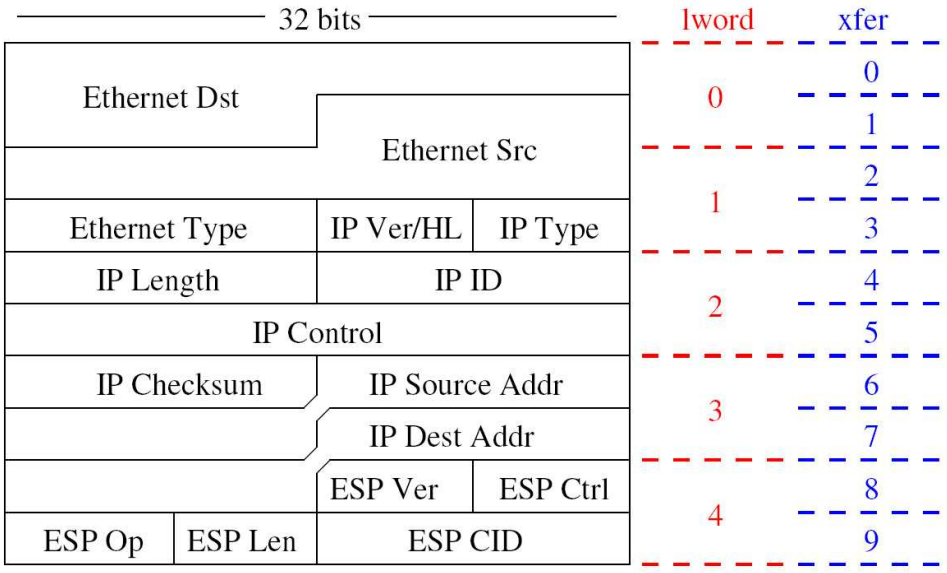
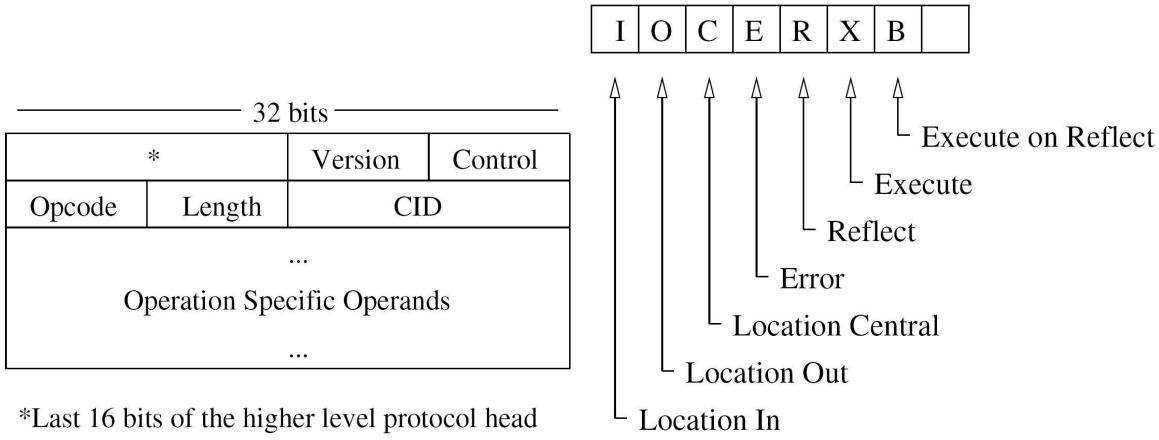
ESP Packet and ESP Instruction Specification

- **ESP Packet Format**





The Execute bit tells an ESP enabled system whether the packet is just "passing though" or needs to be processed.

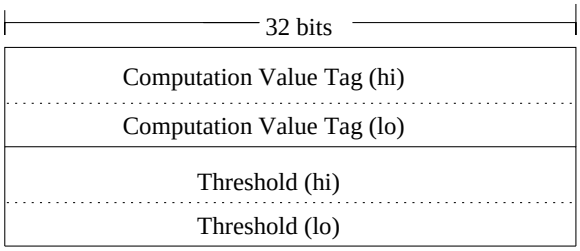
When an end system receives a packet with the Reflect bit set it will set Execute to the value of Execute on Reflect, swap the addresses in the packets and send it off again. In order to execute only on the way back the last 4 bits of the Control byte would be 0xA.

The Error and Execute bits overlap in function but not completely. Its not sufficient to use the Error bit for execution control, e.g. if an error occurs in processing in the outbound direction and the Reflect and Execute on Reflect bits are set the original source will want to see the Error bit regardless of whether Execution was supposed to happen on the path back.

The Length field is the number of 32bit words in the list of operands. Until the alignment of operands relative to the entire packet is not important I don't think there's a simple solution to fitting the length in 8 bits.

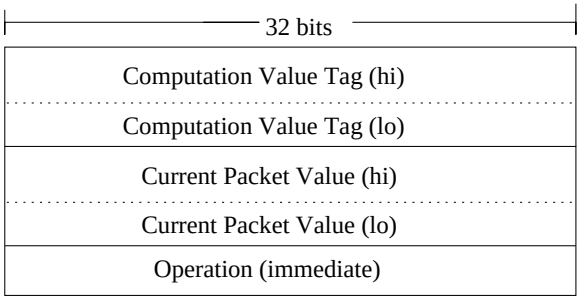
- **COUNT Instruction Format and Pseudocode**

```
count(packet p)
{
    curVal = get (p.compValTag);
    if (curVal == NULL )
        curVal = 0
    curVal = curVal + 1;
    put (p.compValTag, curVal);
    if (curVal <= p.thresh);
        forward p;
    else
        discard p;
}
```



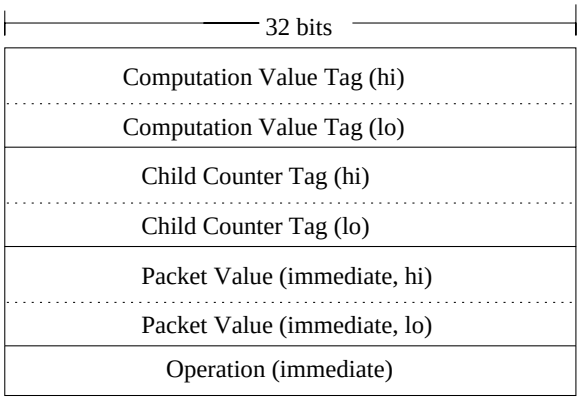
- **COMPARE Instruction Format and Pseudocode**

```
compare(packet p)
{
    curVal = get (p.compValTag);
    if (curVal == NULL){
        put (p.compValTag, p.pktVal);
        forward p;
    }else if (p.op (curVal, p.pktVal)){
        put (p.compValTag, p.pktVal);
        forward p;
    }else
        discard p;
}
```



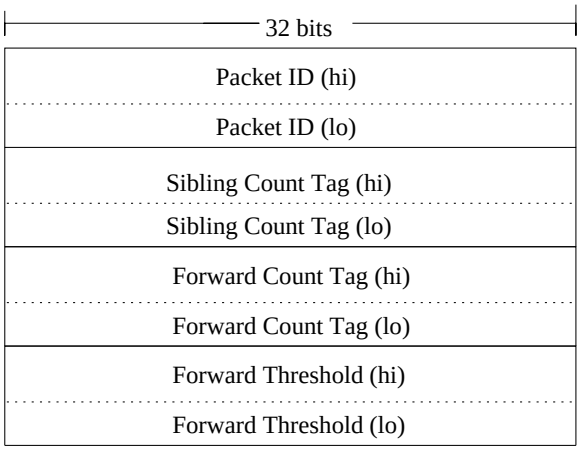
- **COLLECT Instruction Format and Pseudocode**

```
collect (packet p)
{
    curVal = get (p.compValTag);
    if (curVal == NULL)
        curVal = p.val;
    else
        curVal = p.op (curVal, p.val);
    put (p.compValTag, curVal);
    chldCnt = get (p.chldCntTag);
    if (chldCnt == NULL)
        abort;
    chldCnt = chldCnt - 1;
    put (p.chldCntTag, chldCnt);
    if (chldCnt == 0){
        p.val = curVal;
        forward p;
    }else
        discard p;
}
```



- **RCHLD Instruction Format and Pseudocode**

```
rchld (packet p)
{
    sibCnt = get (p.sibCntTag);
    if (sibCnt == NULL)
        sibCnt = 0;
    pktIdSeen = get (p.pktId);
    if (pktIdSeen == NULL){
        pktIdSeen = 1;
        put (p.pktId, pktIdSeen);
        sibCnt = sibCnt + 1;
        put (p.sibCntTag, sibCnt);
    }
    fwdCnt = get (p.fwdCntTag);
    if (fwdCnt == NULL)
        fwdCnt = 0;
    fwdCnt = fwdCnt + 1;
    put (p.fwdCntTag, fwdCnt);
    if (fwdCnt <= p.fwdThresh){
        p.pktId = NODE_ID;
        forward p;
    }else
        discard p;
}
```



- **RCOLLECT Instruction Format and Pseudocode**

```
rcollect(packet p)
{
    pktIdSeen = get(p.pktId);
    if (pktIdSeen == NULL)
        abort;
    sibCnt = get(p.sibCntTag);
    if (sibCnt == NULL)
        abort;
    if (pktIdSeen == 1){
        pktIdSeen = 0;
        put(p.pktId, pktIdSeen);
        sibCnt = sibCnt - 1;
        put (p.sibCntTag, sibCnt);
        compTot = get (p.compId);
        if (compTot == NULL)
            compTot = p.val;
        else
            compTot = p.op (compTot, p.val);
        put(p.compId, compTot)
    }else
        compTot = get (p.compId);
    if (sibCnt == 0){
        fwdCnt = get (p.forwardCount)
        if (fwdCnt == NULL)
            fwdCnt = 0;
        fwdCnt = fwdCnt + 1;
        put (p.forwardCount, fwdCnt);
        if (fwdCnt <= p.fwdThresh){
            p.pktId = NODE_ID;
            p.val = compTot;
            forward p;
        } else
            discard p;
    }else
        discard p;
}
```