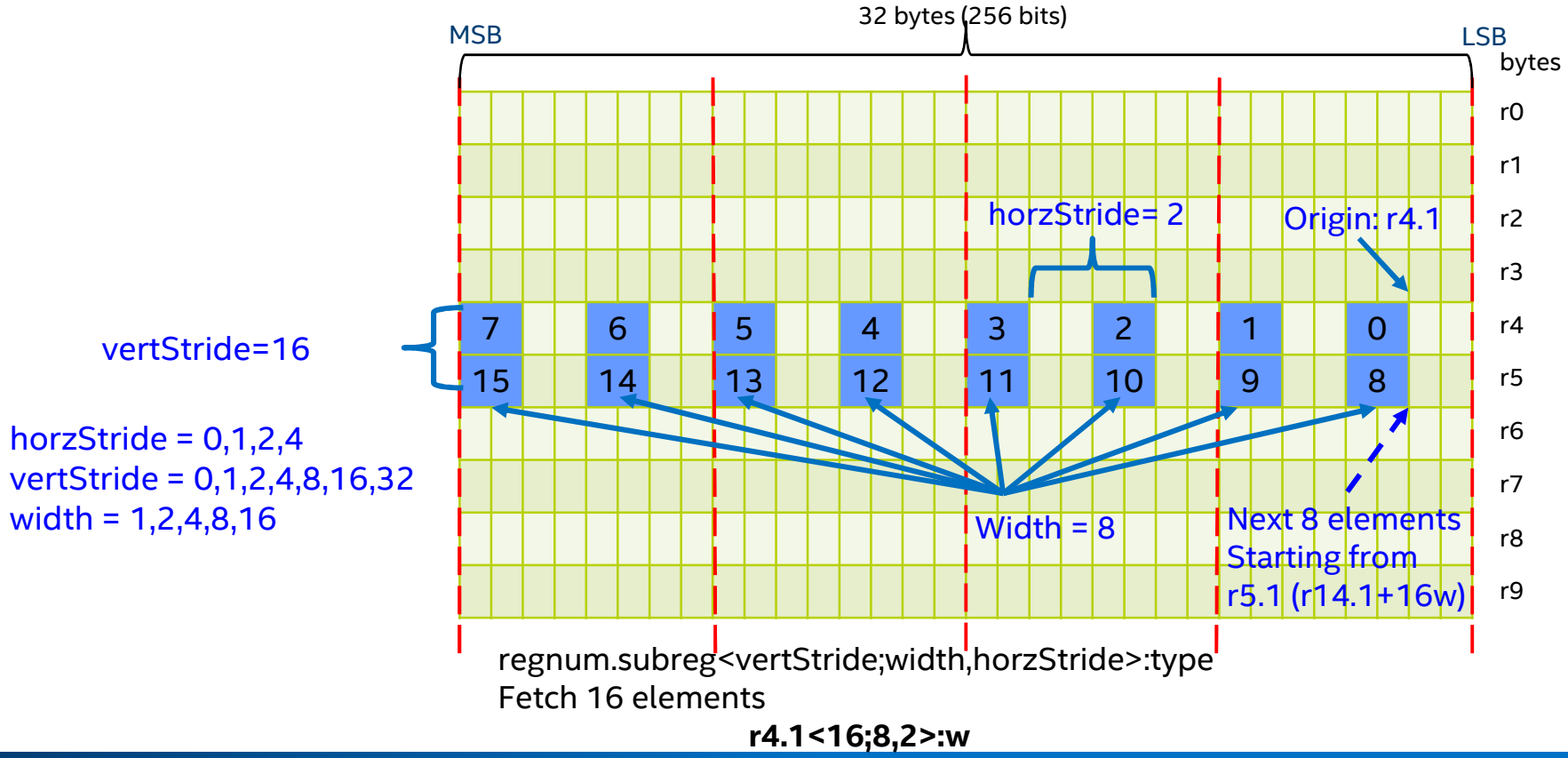
INTEL® GRAPHICS ISA

Ken Lueh – Intel Sr. Principal Engineer & Compiler Architect

Intel Visual & Parallel Computing Group

Source Register region



Source register region

r0.2<1;1,0>:f

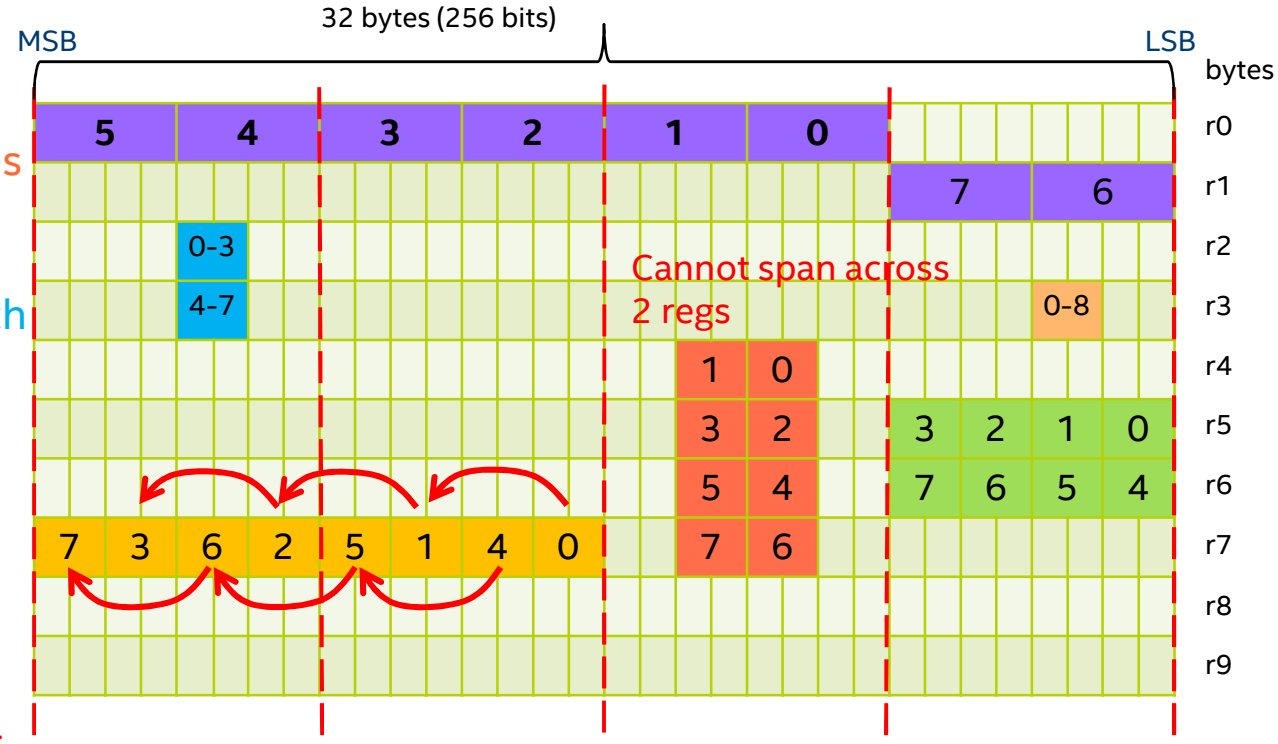
// replicate scalar 8 times
r3.1<0;8,0>:w

// replicated 4 times each
r2.13<16;4,0>:w

// a 2x4 block
r5.0<16;4,1>:w

r7.8<1;4,2>:w

r4.5<16;2,1>:w // illegal



Fetch 8 elements

Destination register region

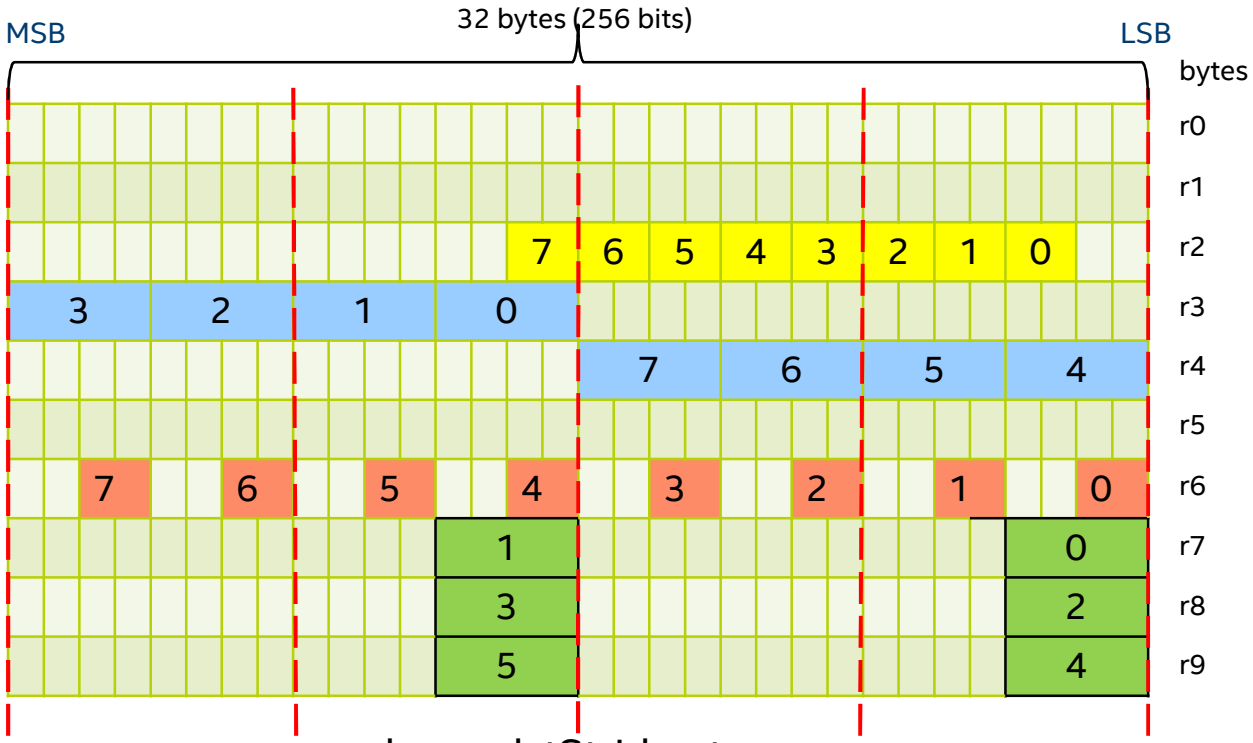
r2.1<1>:w

r3.5<1>:f

r6.0<2>:w

r7.0<4>:f // illegal
// span across 2 regs

dstStride = 1,2,4

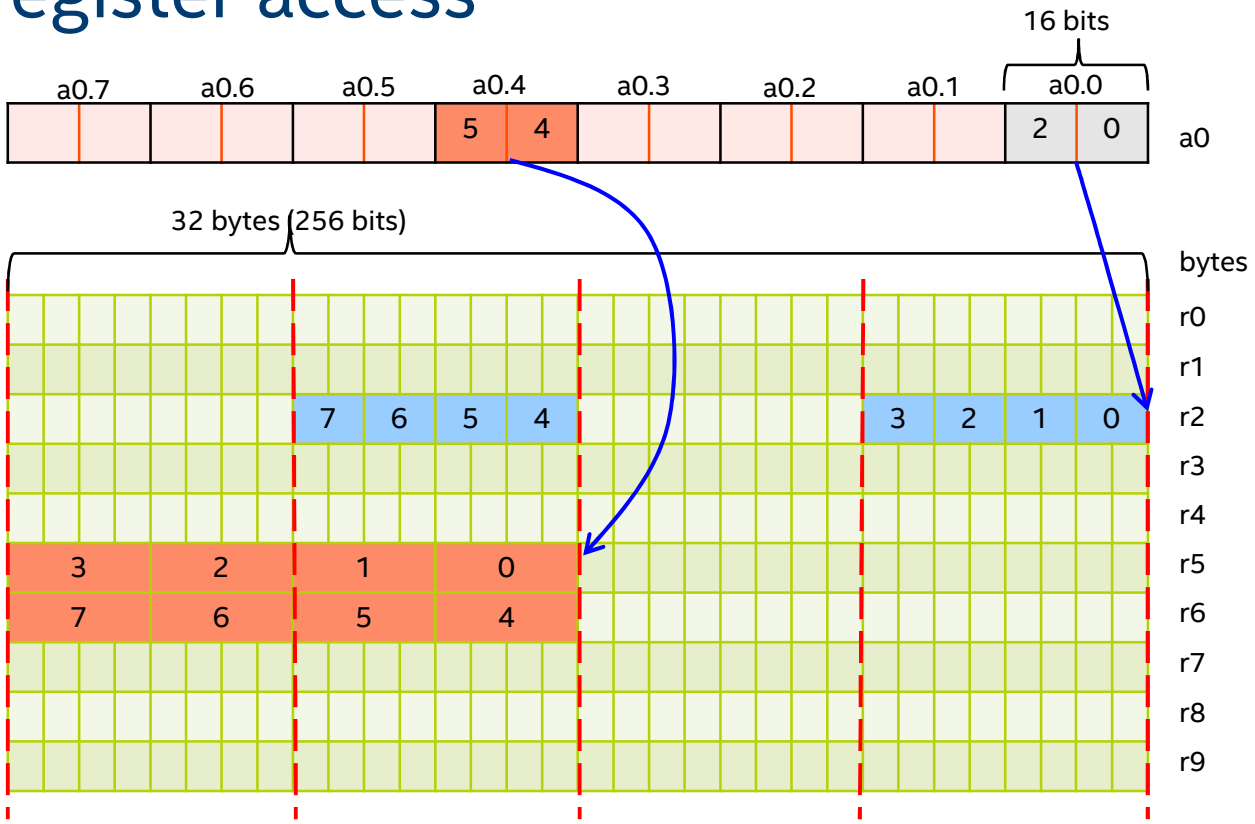


regnum.subreg<dstStride>:type
Write 8 elements

1x1 Indirect register access

r[a0.0]<8;4,1>:w

r[a0.4]<8;4,1>:f



Fetch 8 elements

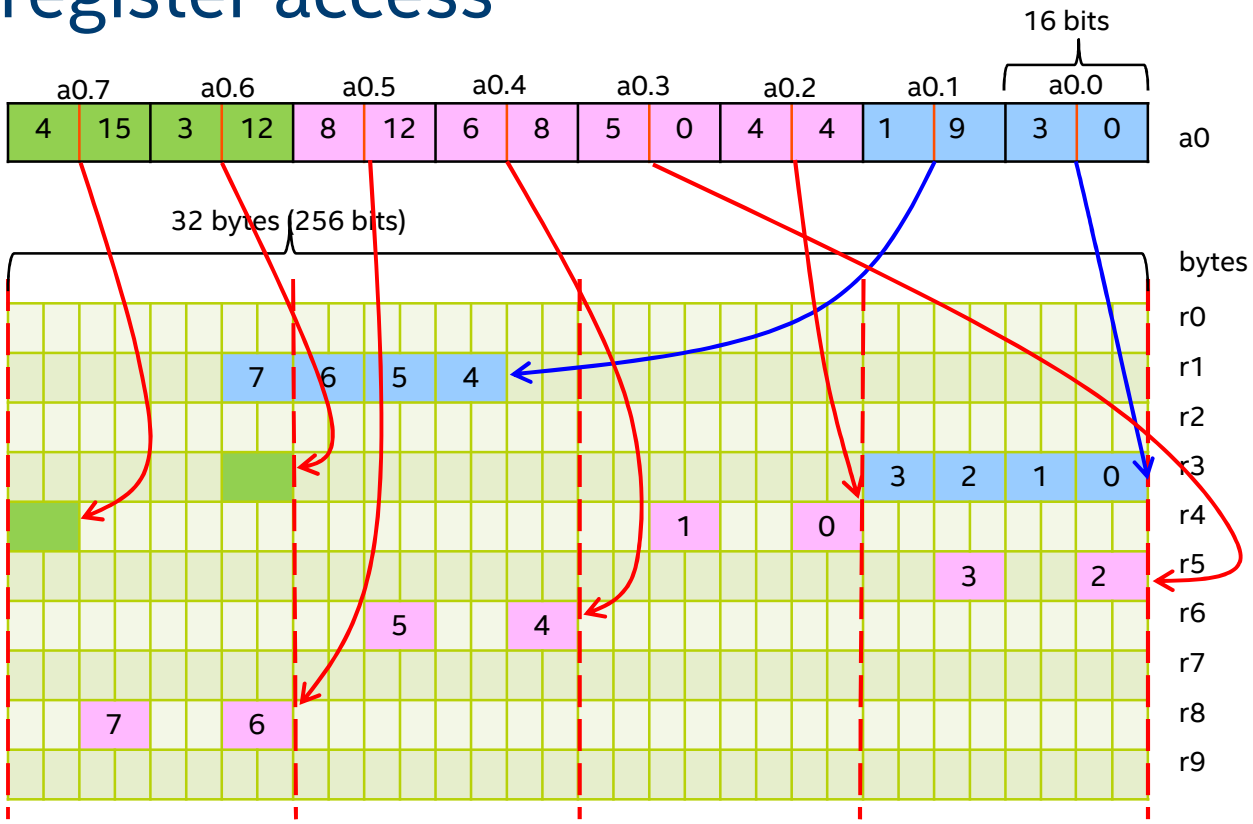
Vx1 Indirect register access

r[a0.0]<4,1>:w

r[a0.2]<2,2>:w

// 8 different locations
r[a0.0]<1,0>:w

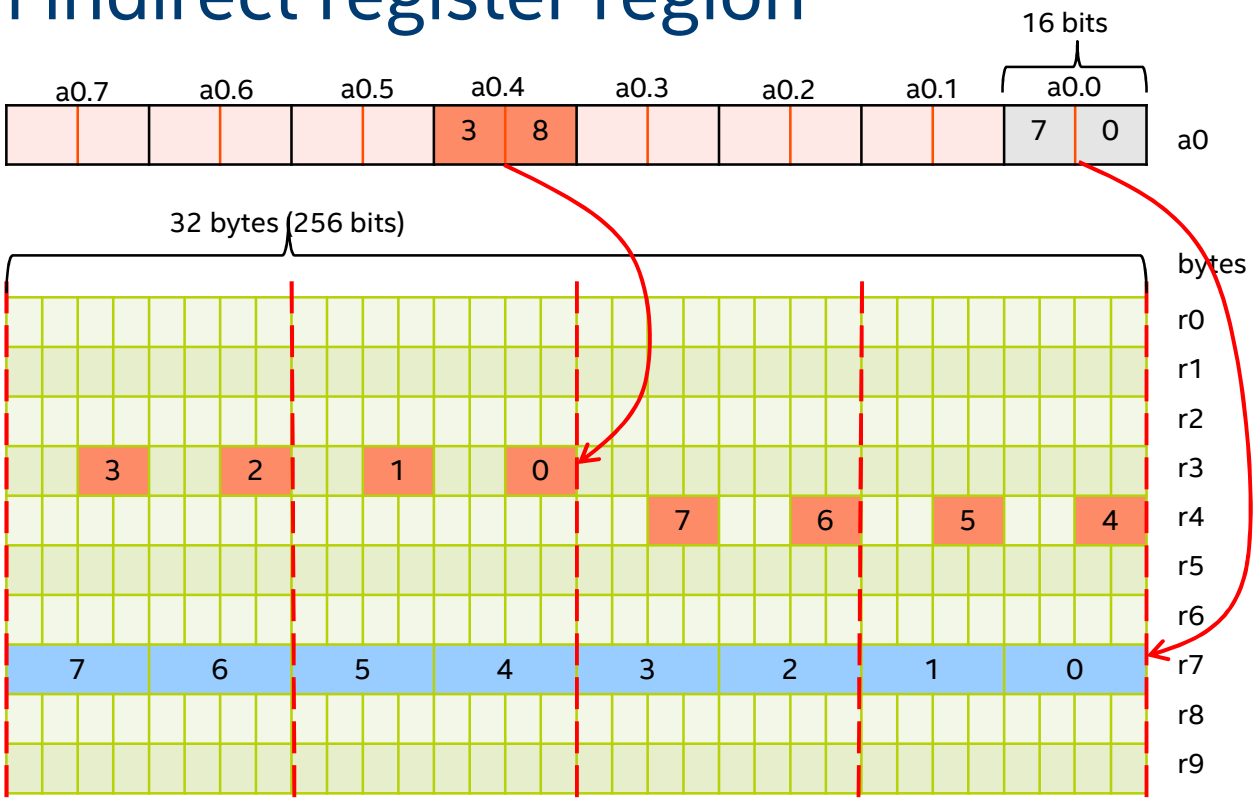
Fetch 8 elements



Destination indirect register region

r[a0.0]<1>:f

r[a0.4]<2>:w



Write 8 elements

Gen ISA overview

```
(f0.0) add (16|M0) [(lt)f0.0] (sat)r10.0<1>:f -r12.0<0;1,0>:f  3.14159:f
```

Predication

- Selectively disables individual lanes for a single instruction based on flag register
- Predication and-ed with execution mask

Execution mask

- Implicitly predicates instruction execution (can override and make all channels execute)

Flag (condition) modifier

- Like flag register in CPU, but we must explicitly set it

Saturation / Source Modifiers

- Saturation clamps arithmetic to destination type
- Source modifier: negation or absolute value of src input

Instruction opcodes

Arithmetic

- ADD, MUL, ASR, MAD, MAC, DP2, DP3, DP4, . . .

Logic

- AND, OR, XOR, NOT, SHR, SHL

Math unit

- Opcode: INV, LOG, EXP, SQRT, RSQ, POW, SIN, COS, INT DIV

Control flow

- IF, ELSE, ENDIF, WHILE, CONTINUE, BREAK, JMPI, RET

Message

- SEND // sampler, load/store, atomic

MISC

- MOV, SEL, PLN, WAIT, FRC, . . .

Instruction opcodes

AVG: Integer average of two source operands, rounding up
BFE: Extract a bit field defined by src0 and src1 from src2
BFI1: Bit field insertion
BFI2: Bit field insertion
BFREV: Reverse all bits in src0
CBIT: Count the number of bits set (1) in src0
FBH: First significant bit searching from the high bits in src0
FBL: First significant bit searching from the low bits in src0
LINE: Compute a line equation (v = p * u + q) of src0 and src1
LRP: Compute the linear interpolation equation
LZD: Count the number of leading 0 bits (from MSB) in src0
PLN: Compute a plane equation (w = p*u + q*v + r)
RNDD, RNDE, RNDU, RNDZ: rounding
SAD2, SADA2: two-wide sum-of-absolute-difference operation

Execution mask

32-bit SIMD execution mask enables selective channels

Controls which part of ARF registers instructions use

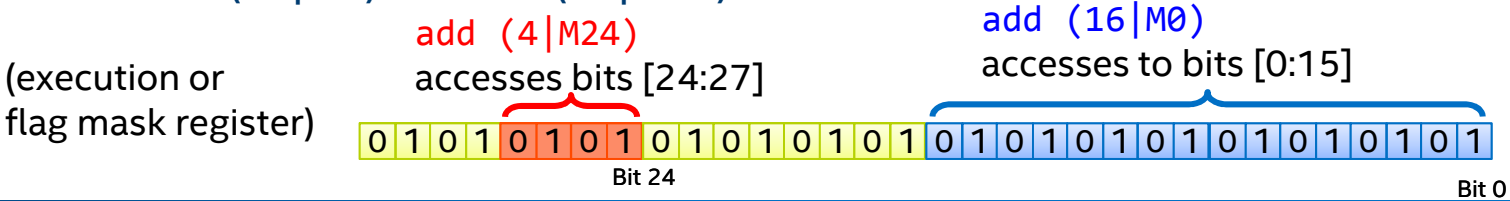
Control flow changes the execution mask

(exec_size | mask_offset)

- exec_size: 1, 2, 4, 8, 16, 32
- mask_offset: bit position, mask_offset%4 == 0

Caveat: not all combinations are legal

- add (16 | M24) --- out of bound
- add (32 | M0) --- for float or int, not legal because HW does not allow: broken into add (16|M0) and add (16|M16)



Predication

Uses a flag register to selectively deactivate channels

- Two 32-bit flag registers, f0 and f1

Helps eliminate control flow

~ complements (inverts) the flag register value first

Uses as many bits from flag register as execution width

- Execution offset tells where in the flag register to start reading from

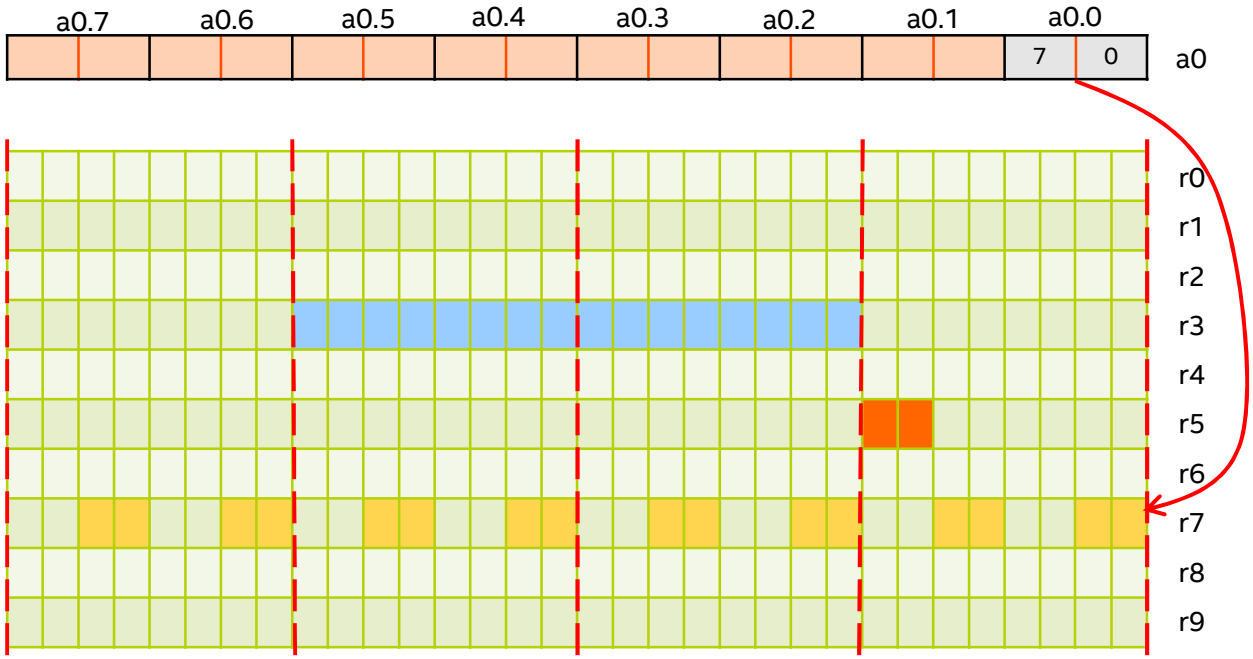
```
(f1.0)    mov  (8|M0)  r2.0<1>:f  3.14:f
(~f1.0)   mov  (8|M0)  r2.0<1>:f  2.71:f
```

Flag (condition) modifier

`[(lt)f0.0]`

- **Computed result is compared against zero**
- **Compared result is set in flag register**
- **Compare opcode: eq, ne, gt, ge, lt, le, o, and u**
  - o: tests whether the computed result causes overflow
  - u: tests whether the computed result is a NaN

Example




```
add (8|M8) [(e)f0.0] (sat)r[a0.0]<2>:w r3.4<8;8,1>:w -r5.3<0;1,0>:w
```

Send message

Perform data communication between a HW thread and external function units

- Sampler, Data Port Read, Data Port Write, URB and some fixed functions

Add an entry to the EU's message request queue

The response message, if present, will be returned to a block of contiguous GRF registers

Two types of send

- Send – has one payload source

  Dst **r10 r11 r12 r13** src0 **r20 r21 r22 r23 r24 r25 r26 r27**

- Sends – has two payload sources

  Dst **r10 r11 r12 r13** src0 **r20 r21 r22 r23** src1 **r30 r31 r32 r33**

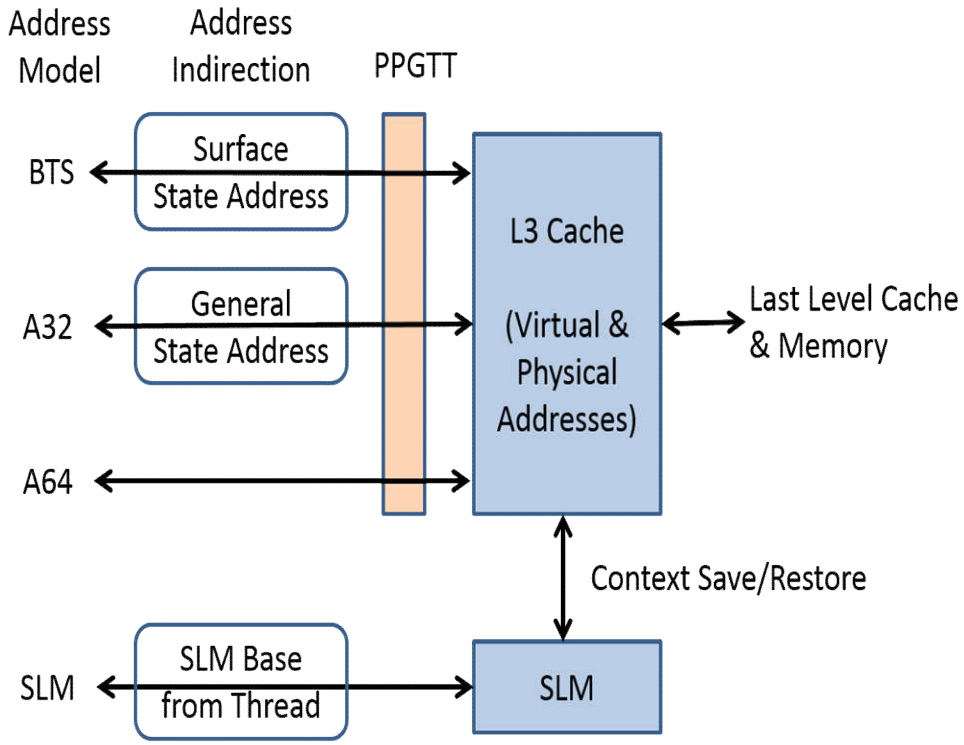
Sends example

sendType	destination	src0	src1	extMsgDesc	msgDesc
sends	(8|M0)	r4:ud	r8	r9	0x42	0x24AA019

- **Extended Message Descriptor (extMsgDesc)**
  - Bits:0x42 →0001 0 0 0010:b
    - [3,0] Shared Function ID - Sample: 2
    - [6,9] Src1 message Length: 1GRF
- **Message Descriptor (msgDesc)**
  - Bits: 24AA019 →0001 00100 1 01 01010 0000 00011001:b
    - [7,0] Binding Table Index: 25
    - [11,8] Sample Index: 0
    - [16,12] Message Type - ResInfo: 10
    - [18,17] SIMD Mode – SIMD8 : 1
    - [19] Header Present – True: 1
    - [24,20] Response Length: 4GRF
    - [28,25] Src0 Message Length: 1GRF

Address modes

- BTS (Binding Table State Model):
  - Binding table entry contains a pointer to the surface state
  - Address = Surface state address + offset
  - Used for sampler, typed surfaces, etc.
- A32 Stateless (BTI=253/255)
  - Address = General state base address + offset
  - Used for scratch space, private memory, and 32-bit global memory access
- A64 Stateless (BTI=253/255)
  - Address = offset
  - Used for SVM and 64-bit global memory access
- SLM (BTI=254)
  - Address = SLM thread base + offset
  - Used for local memory access



DataPort messages

Message	Data type	SIMD Size	Data Elements	Description
Scattered R/W	B,DW,QW	8,16	1,2,4	Read or write 8/16 scatter B/W/DW starting at each offset
Block R/W	OW	1	1	Read or write 1,2,4,8 contiguous Owords starting at the offset
Untyped Surface R/W	DW	8,16	1,2,3,4	Read or write up to 4 DWords at each of the 8/16 offsets to an untyped surface
Typed Surface R/W	DW	8	1,2,3,4	Read or write up to 4 DWords at each of the 8 offsets to an typed surface
Media Block R/W	DW	1	Width * Height	Read or write a rectangular block of data starting at coordinate (X,Y)
Untyped Atomic Messages	DW	8,16	1	Atomic read-modify-write at 8/16 offsets

- Selected examples of memory access messages
  - Not all messages support all address models
  - Various platform-specific restrictions -- consult Gen spec for details

Structured control flow - IF

[(pred)] if (exec_size) JIP UIP

```
for ( n = 0; n < exec_size; n++ ) {
   if ( WrEn.channel[n] == 0 ) {
      PcIP[n] = IP + JIP;   // channels not entering the IF-block
   }
   else {
      PcIP[n] = IP + 1;     // channels entering the IF-block
   }
}
if ( PcIP != (IP + 1) ) { // for all channels
   Jump(IP + JIP);     // if none of channels enters the if-block, skip the block
}
```

Structured control flow - ELSE

else (exec_size) JIP UIP

```
for (n = 0; n < exec_size; n++) {
   if (WrEn.channel[n] == 1) {
      PcIP[n] = IP + 1;     // channels entering the ELSE block
   }
   else {
      PcIP[n] = IP + JIP;   // channels not entering the ELSE  block;
   }
}
if (PcIP != (IP+1)) { // for all channels
   Jump(IP + JIP);  // none of channels enters the ELSE block; skip the  block
}
```

Structured control flow - ENDIF

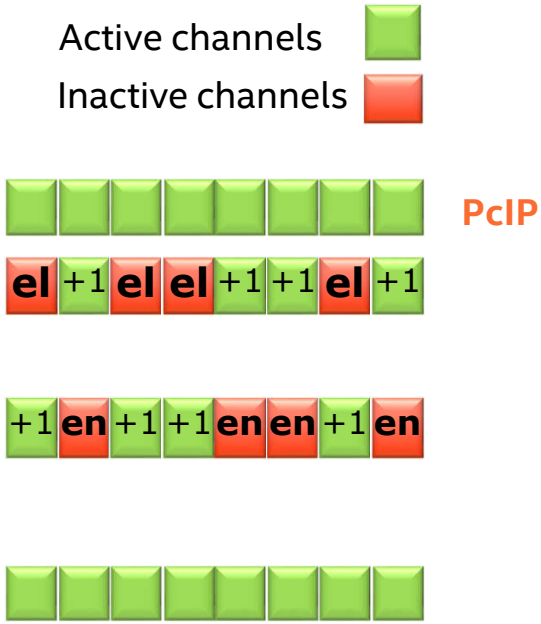
endif (exec_size) JIP

```
if ( WrEn == 0 ) { // all channels false
   Jump(IP + JIP);
}
```

```
cmp (8|M0)  [(ne)f0.0] . . .
(f0.0) if  (8)  off_endif  off_else
  . . .

else (8|M0)  off_endif
  . . .

endif  (8|M0)  off_redirect
```

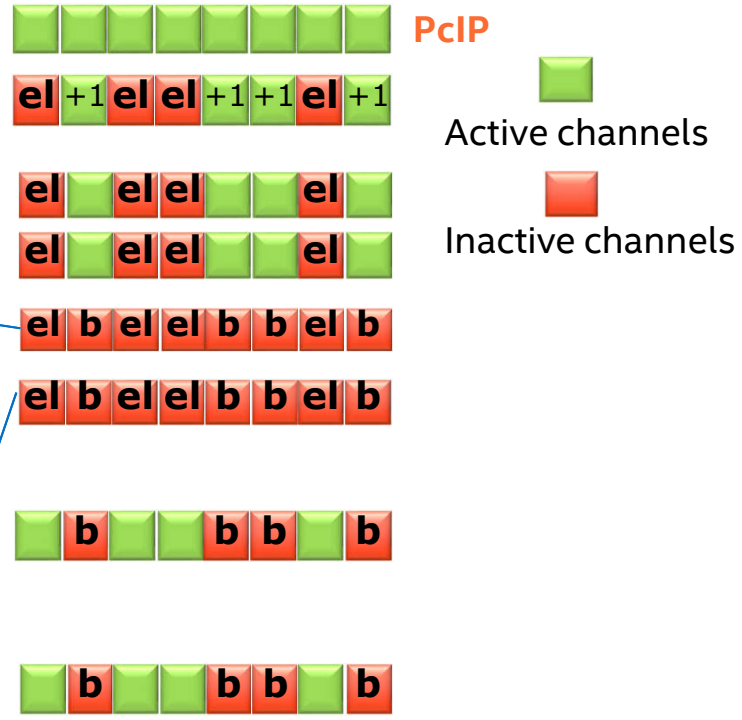


el: IP of else
en: IP of endif

Structured control flow – nested IF example

```
cmp (8|M0) [(ne)f0.0]. . .
(f0.0) if (8|M0)  off_endif1 off_else
    . . .
    cmp (8|M0) [(g)f0.0]. . .
    (f0.0) if (8|M0) off_endif2 off_endif2
         . . .
         break off_endif2  off_br
    endif (8|M0)  off_else // endif2
    ...
else (8|M0)  off_endif
    . . .

endif  (8|M0)  off_redirect  // endif1
```



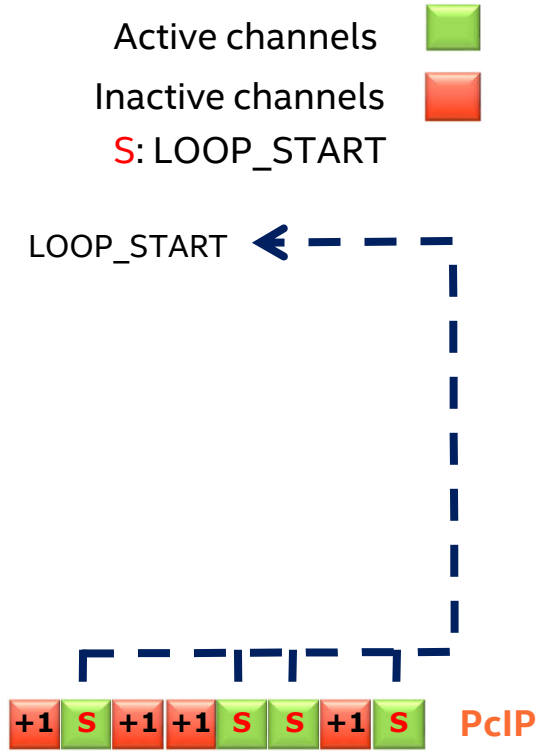
Structured control flow - WHILE

[(pred)] while (exec_size) JIP

```
for ( n = 0; n < 32; n++ ) {
   if (WrEn.chan[n] ) {
      PcIP[n] = IP + JIP;  // channels that continue the while loop
   }
   else {
      PcIP[n] = IP + 1;   // channels that exit the while loop
   }
}
if ( PMask == 1 ) { // any enabled channel true
   Jump(IP + JIP);
}
```

```
cmp (8|M0) [(gt)f0.0]
```

(f0.0) while (8|M0) LOOP_START



Structured control flow - BREAK

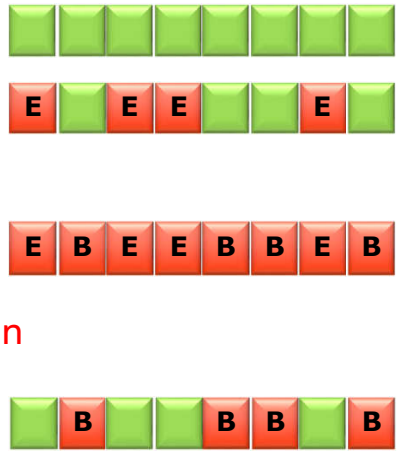
Early-out from the inner most loop, or
break instruction terminates the loop for all execution channels enabled

```
[(pred)] break (exec_size) JIP UIP
for ( n = 0; n < exec_size; n++ ) {
   if ( WrEn.channel[n] ) {
      PcIP[n] = IP + UIP;    // channels break out of loop
   }
   else {
      PcIP[n] = IP + 1;     // fall through channels
   }
}
if ( PcIP != (IP + 1) ) { // all channels (no channel executes next inst)
   Jump(IP + JIP);   // jumps out of encompassed control flow block
}
```

BREAK example

```
        cmp (8|M0) [(g)f0.0] . . .
(f0.0) if (8|M0)   . . .
           . . .
           break (8|M0) End_label Loop_break
                 ...
End_label:
        endif
        . . .
        cmp (8|M0) [(g)f0.0] . . .
Loop_break:
(f0.0)while (8|M0) Loop_start
```

// skip execution of this section



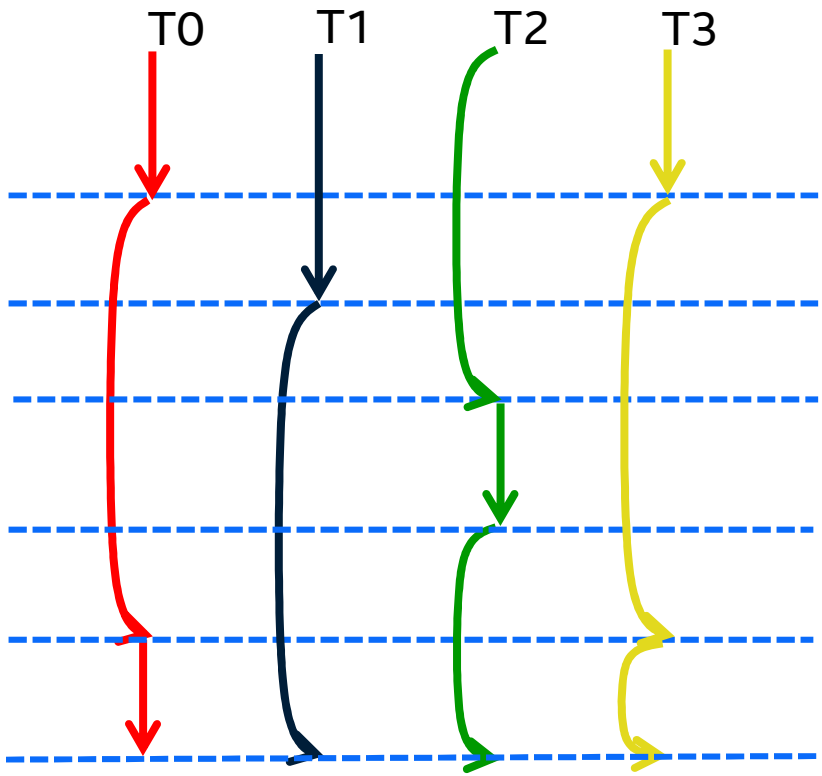
Active channels

Inactive channels

E: IP of end_label
B: IP of Loop_break

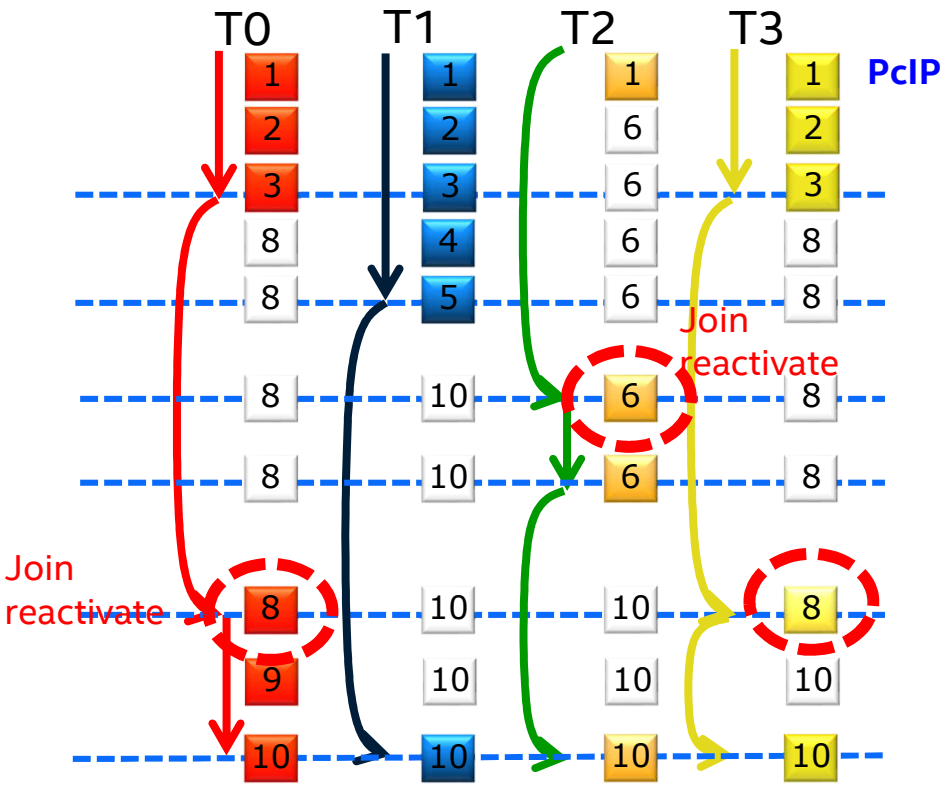
Unstructured control flow -- goto

```
PC
1: If a > b goto Label_A
2:    code_1
3: if c > d goto Label_B
4:    code_2
5:    goto  Label_C
Label_A:
6:    code_3
7:    goto Lable_C
Label_B:
8: if e == 0 goto Label_C
9:    code_4
Label_C:
10:
```



Unstructured control flow -- goto

```
PC
1: If a > b goto Label_A
2:    code_1
3: if c > d goto Label_B
4:    code_2
5:    goto  Label_C
Label_A:
6:    code_3
7:    goto Lable_C
Label_B:
8: if e == 0 goto Label_C
9:    code_4
Label_C:
10:
```

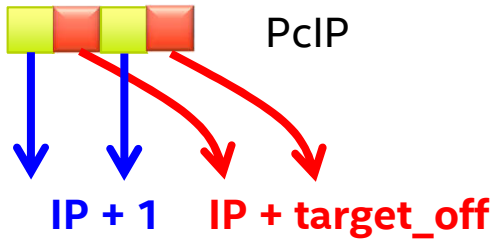


goto

(pred) goto (exec_size) join_off target_off <brch_ctrl>
When <brch_ctrl> is false

```
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n])
        // for the predicated active channels
        pcip[n] = IP + 1; // fall through channels
    else
        // join ip, for the active non predicated channels
        pcip[n] = IP + target_off;
}

if (pcip != IP + 1)     // for all channels
    jump(ip + join_off);  // jump to join point
Else // at least one channel falls through
    jump(ip + 1);
```

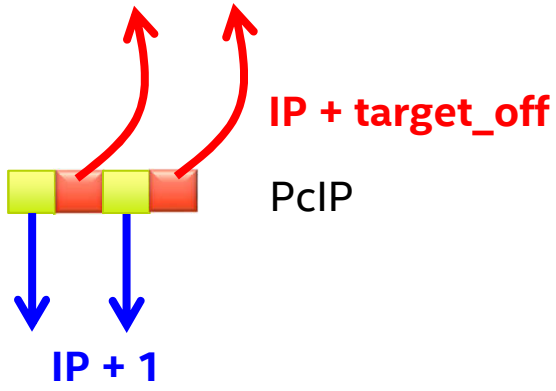


goto

(pred) goto (exec_size) join_off target_off <brch_ctrl>

When <brch_ctrl> is true

```
for (n = 0; n < exec_size; n++) {
    if (WrEn.chan[n])
        pcip[n] = IP + target_off; // for the predicated active channels
    else
        // join ip, for the active non predicated channels
        pcip[n] = IP + 1;
}
if (pcip != IP + target_off) ) {    // for all channels
    if (pcip != IP + 1) ) {  // for all channels
        jump(ip + join_off); // jump to join point
    else
        jump(ip + 1);
} else {
    // at least one channel to target
    jump(ip + target_off);
}
```

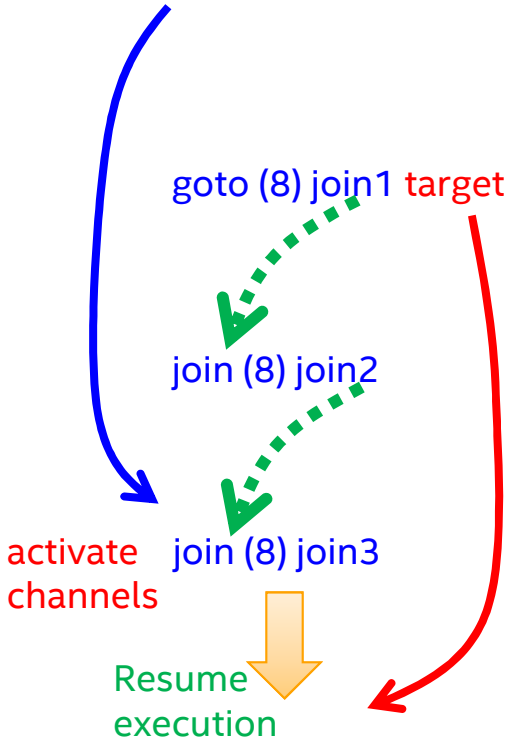


Join

(pred) join (exec_size) JIP

- **Join instruction is used in conjunction with goto**
- **Any deactivated channels due to goto match join IP are activated**

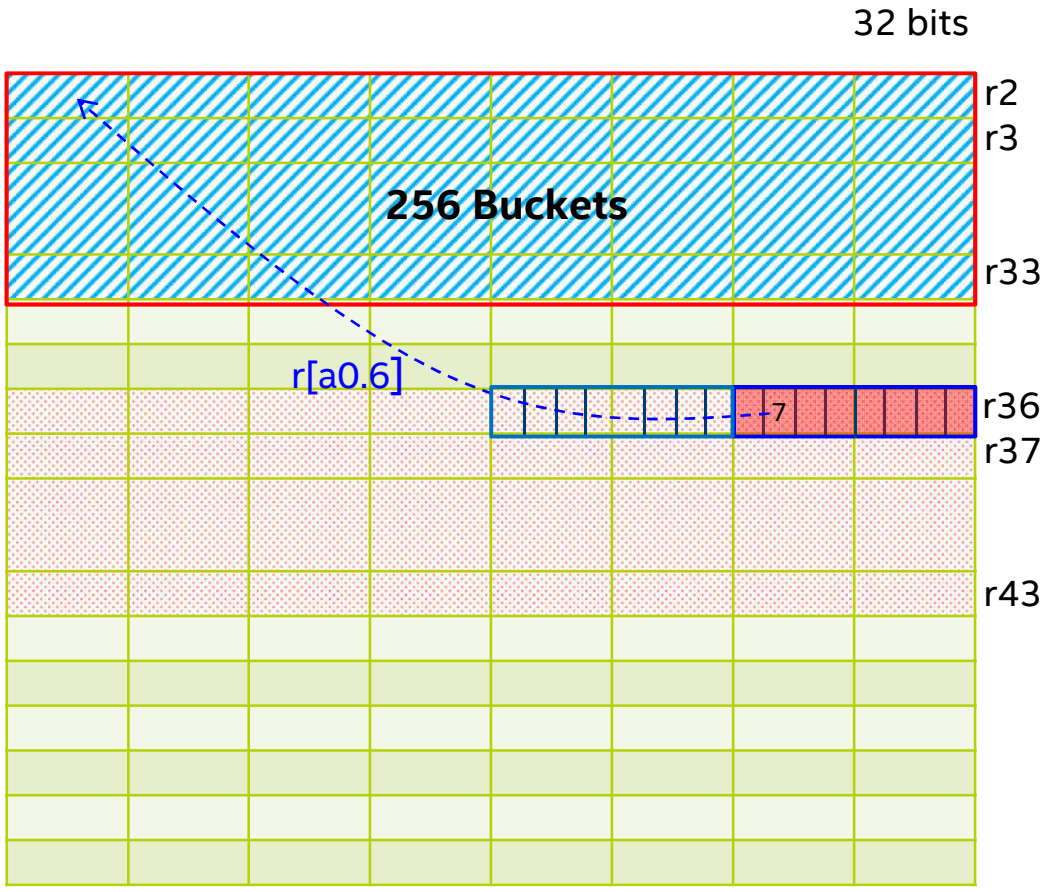
```
for ( n = 0; n < exec_size; n++ ) {
    if (WrEn.chan[n] ) {
       PcIP[n] = IP + 1;
    }
}
// for all channels when no channel is
// activated
if ( PcIP != (IP + 1) ) {
   Jump(IP + JIP); // skip to next join
}
```



Example: Histogram

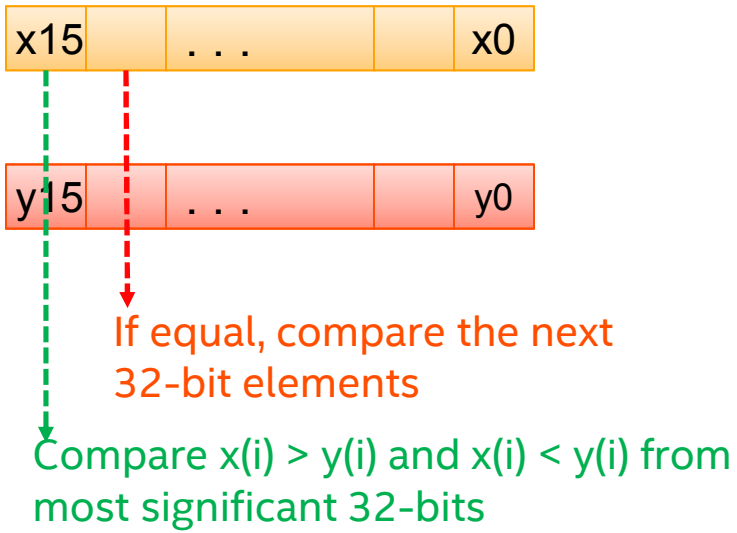
```
// init histogram buckets
mov (16) r2.0<1>:d 0:w
...
mov (16) r32.0<1>:d 0:w

// read  8x32 bytes from image
send (8|M8) r36.0<1>:ud . . .
// for (I = 0; I < 8; i++)
//     for (j = 0; j < 32; j++) {
//         histogram(in(i,j)) += 1;
mul (8|M0) r44.0<1>:uw r36.0<8;8,1>:ub 0x4:uw
add (8|M0) a0.0<1>:uw r44.0<8;8,1>:uw 0x40:uw // &r2
add (1|M0) r[a0.0, 0]<1>:d r[a0.0, 0]<0;1,0>:d 0x1:w
add (1|M0) r[a0.1, 0]<1>:d r[a0.1, 0]<0;1,0>:d 0x1:w
...
add (1|M0) r[a0.6, 0]<1>:d r[a0.6, 0]<0;1,0>:d 0x1:w
add (1|M0) r[a0.7, 0]<1>:d r[a0.7, 0]<0;1,0>:d 0x1:w
 // process the next 8 bytes
mul (8|M0) r45.0<1>:uw r36.8<8;8,1>:ub 0x4:uw
add (8|M0) a0.0<1>:uw r45.0<8;8,1>:uw 0x40:uw
add (1|M0) r[a0.0, 0]<1>:d r[a0.0, 0]<0;1,0>:d 0x1:w
```



Eample: Compare big integers

```
Int cmp_ge_n(
   vector<u32, 16> x,
   vector<u32, 16> y)
   for(i=15; i >= 0; --i)    {
      if(x(i) > y(i))
            return 1;
      else if(x(i) < y(i))
            return 0;
   }
   return 1; // equal
}
```

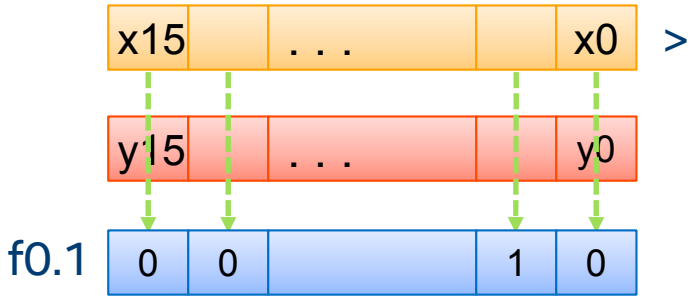


SIMD fashion

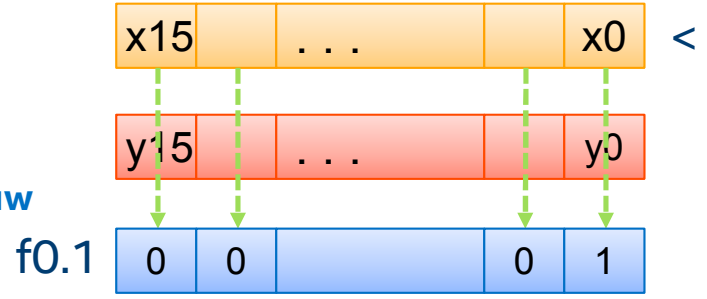
```
//  16 element comparison x > y
cmp (16|M0) [(g)f0.1] null<1>:d r3.0<8;8,1>:ud r5.0<8;8,1>:ud
mov (1|M0) r1.1<1>:uw f0.1<0;1,0>:uw

// 16 element comparison x < y
cmp (16|M0) [(l)f0.0] null<1>:d r3.0<8;8,1>:ud r5.0<8;8,1>:ud
mov (1|M0) r1.0<1>:uw f0.0<0;1,0>:uw

// return r1.1  > r1.0;
cmp (1|M0) [(ge)f1.0] null<1>:uw r1.1<0;1,0>:uw r1.0<0;1,0>:uw
mov (1|M0) r1.0<1>:w 0x1:w
(f1.0) sel (1|M0) r1.0<1>:w r1.0<0;1,0>:w 0:w
```

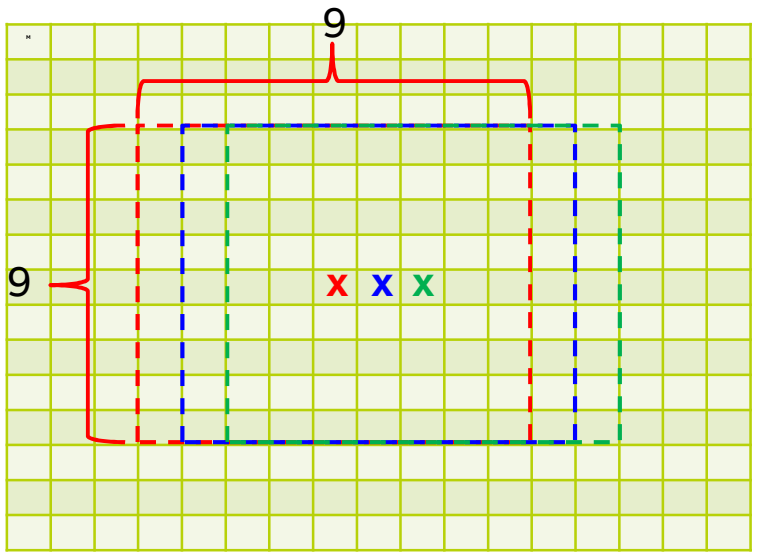


R1.1 = 001...10b // greater

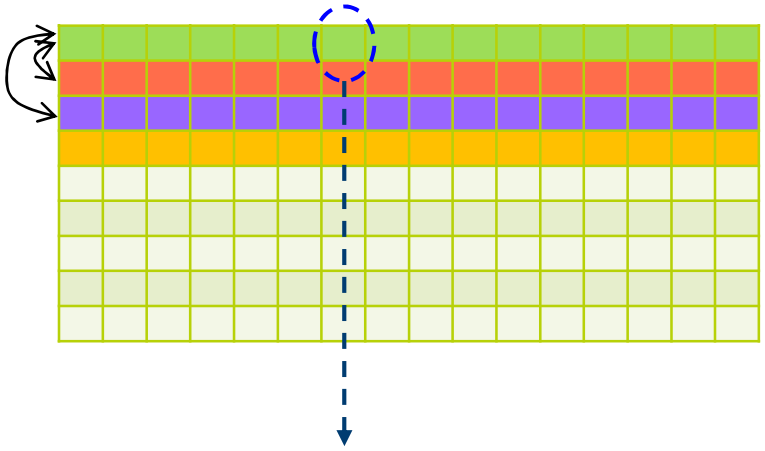


R1.0 = 000...01b // less

Example: 9x9 Min/Max filter



Compute Min/Max for each column

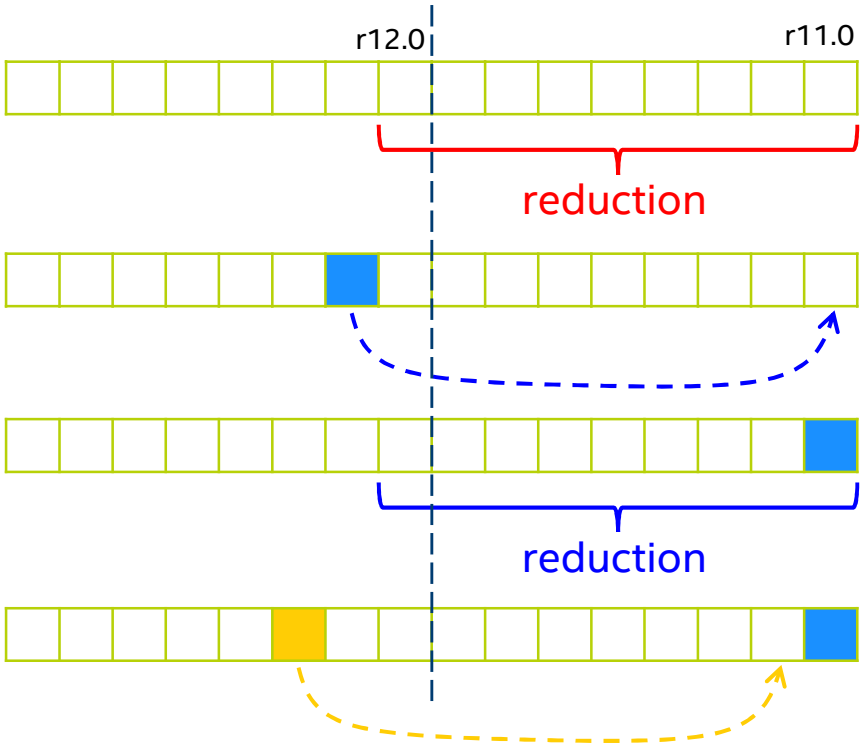



```
// element-wise select max between first row and second row
// keep max in the first row
sel.ge.f0.0 (16) r11.0<1>:f r13.0<8;8,1>:f   r11.0<8;8,1>:f

// element-wise select max between the current max (first row)
// and third row
sel.ge.f0.0 (16) r11.0<1>:f r15.0<8;8,1>:f   r11.0<8;8,1>:f

// element-wise select max between the current max (first row)
// and fourth row
sel.ge.f0.0 (16) r11.0<1>:f r17.0<8;8,1>:f   r11.0<8;8,1>:f
```

Compute 9x9 min/max




```
// compute 9x9 max for the first pixel (r72.0)
sel (4|M0) [(ge)f0.0] r69.0<1>:f   r11.0<4;4,1>:f   r11.4<4;4,1>:f
sel (2|M0) [(ge)f0.0] r70.0<1>:f   r69.0<2;2,1>:f   r69.2<2;2,1>:f
sel (1|M0) [(ge)f0.0] r71.0<1>:f   r70.0<0;1,0>:f   r70.1<0;1,0>:f
sel (1|M0) [(ge)f0.0] r72.0<1>:f   r71.0<0;1,0>:f   r12.0<0;1,0>:f

// replace the first element (sliding window to the left)
mov (1|M0)  r11.0<1>:f    r12.1<0;1,0>:f

// compute 9x9 max for the 2nd pixel (r72.1)
sel (4|M0) [(ge)f0.0)] r80.0<1>:f   r11.0<4;4,1>:f   r11.4<4;4,1>:f
sel (2|M0) [(ge)f0.0)] r81.0<1>:f   r80.0<2;2,1>:f   r80.2<2;2,1>:f
sel (1|M0) [(ge)f0.0)] r82.0<1>:f   r81.0<0;1,0>:f   r81.1<0;1,0>:f
sel (1|M0) [(ge)f0.0)] r72.1<1>:f   r82.0<0;1,0>:f   r12.0<0;1,0>:f

// replace the 2nd element (sliding window to the left)
mov (1|M0) r11.1<1>:f   r12.2<0;1,0>:f
// compute 9x9 max for the 3rd pixel (r72.2)
. . .
```

Legal Notices and Disclaimers

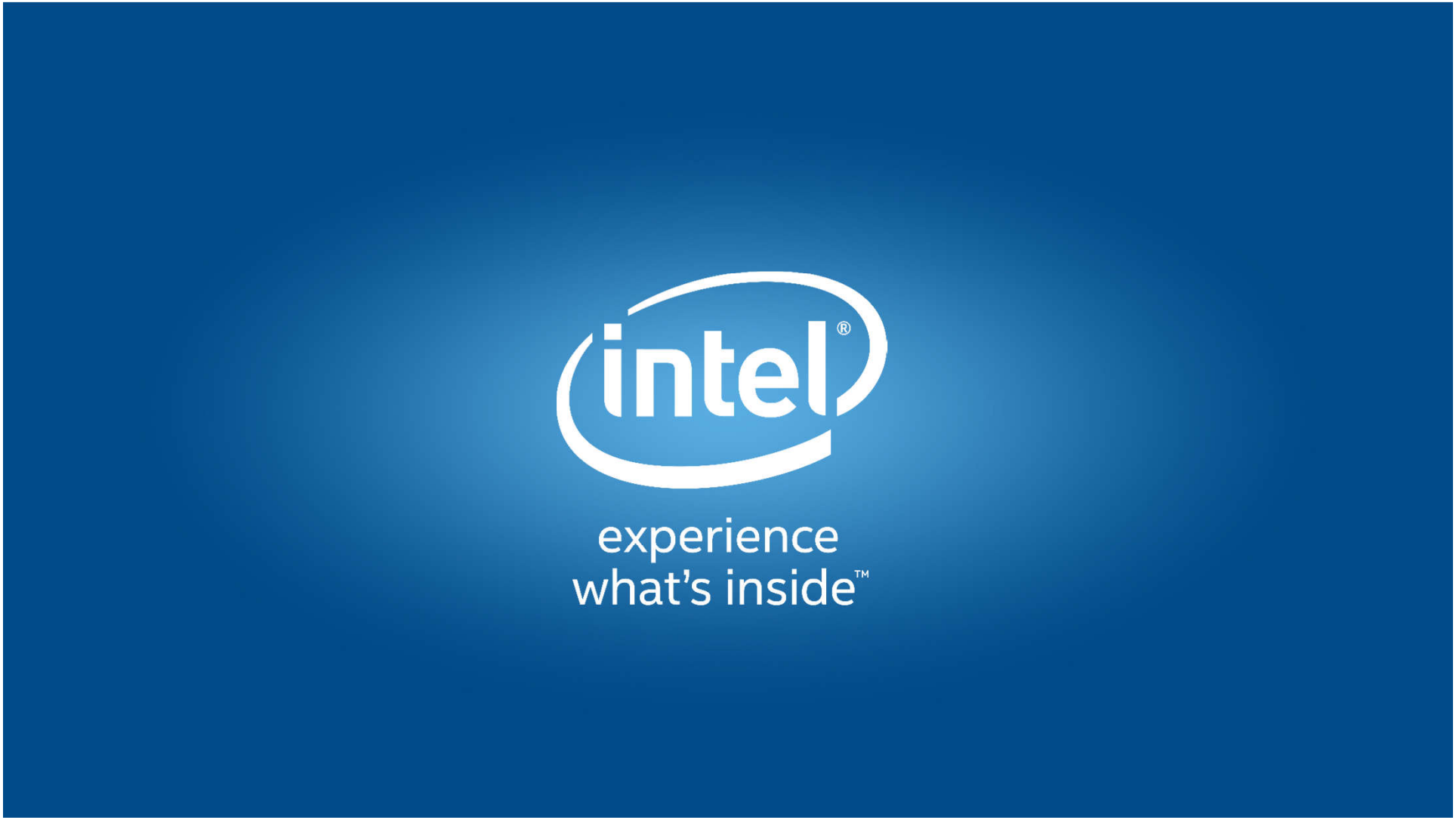
Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

No computer system can be absolutely secure.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit **http://www.intel.com/performance**.

Intel, the Intel logo and others are trademarks of Intel Corporation in the U.S. and/or other countries.
*Other names and brands may be claimed as the property of others.

© 2015 Intel Corporation.


intel®
experience
what's inside™