Bringing ARB_gpu_shader_fp64 to Intel GPUs

Iago Toral Quiroga
<itoral@igalia.com>

XDC 2016
Helsinki, Finland
Contents

→ ARB_gpu_shader_fp64
   → Overview
   → Scope
→ Intel implementation
   → NIR
   → i965
→ Current status
ARB_gpu_shader_fp64

- **New GLSL types**: double, dvecX and dmatX
  - Variables, Inputs, outputs, uniforms, constants (LF suffix)
  - No 64-bit vertex attributes (handled by another extension)
  - No 64-bit fragment shader outputs (no 64-bit FBs)
- **Arithmetic and relational operators** supported
- **Most built-in functions can take the new types**
  - Some exceptions: angle, trigonometry, exponential, noise
- **New packing functions**: (un)packDouble2x32
- **Conversions from/to 32-bit types**
- **No interpolation**
Scope (i965)

- Combined work by Intel and Igalia for over a year
- ~260 new patches to add NIR (~60) and i965 support (~100 Broadwell+, ~100 Haswell)
  - More IvyBridge patches on the way!
- Usable across all shader stages
- 3 IRs to support: NIR, i965/align1, i965/align16
- Lots of GL functionality involved: ALU, varyings, uniforms, UBO, SSBO, shared variables, etc.
- Lots of internal driver modules involved: Optimization passes, liveness analysis, register spilling, etc.
Scope (Piglit)

- Piglit coverage was limited
  - Mostly focused on ALU operations.
  - ~2,000 new tests added
NIR

- Added support for **bit-sized ALU types**:
  - nir_type_float32 = 32 | nir_type_float, etc
  - nir_alu_get_type_size(), nir_alu_get_base_type()
- We need to be a bit more careful now:
  
  ```c
  - if (alu_info.output_type == nir_type_bool) {
  +if (nir_alu_type_get_base_type(alu_info.output_type) == nir_type_bool) {
  ```
NIR

- Lots of plumbing to deal correctly with bit-sizes everywhere.
- Algebraic rules and bit-size validation
- New double-precision opcodes:
  - d2f, d2i, d2u, d2b, f2d, i2d, u2d
  - (un)pack_double_2x32
  - etc
NIR

- Added **lowering for unsupported 64-bit operations on Intel GPU hardware**:
  - `trunc, floor, ceil, fract, round, mod, rcp, sqrt, rsq`
  - `nir_lower_doubles()`. Implemented in terms of:
    - Supported 64-bit operations
    - 32-bit float/integer math
- Drivers can choose which lowerings to use:
  - `nir_lower_doubles_options`
i965 – Fp64
(Align1)
Algin1 – SIMD8 - 32bit

<table>
<thead>
<tr>
<th></th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Row 1
Row 2

\text{mov}(8) \quad g0.0<1>F \quad g1.1<8,4,2>F \quad \{\text{align1} 1Q\}
## Align1 – SIMD8 - 64bit

<table>
<thead>
<tr>
<th>0B</th>
<th>32B</th>
<th>64B</th>
<th>96B</th>
<th>128B</th>
<th>160B</th>
<th>192B</th>
<th>224B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xi</td>
<td>Xi</td>
<td>Yi</td>
<td>Yi</td>
<td>Zi</td>
<td>Zi</td>
<td>Wi</td>
<td>Wi</td>
</tr>
<tr>
<td>Xh</td>
<td>Xh</td>
<td>Yh</td>
<td>Yh</td>
<td>Zh</td>
<td>Zh</td>
<td>Wh</td>
<td>Wh</td>
</tr>
<tr>
<td>Xi</td>
<td>Xi</td>
<td>Yi</td>
<td>Yi</td>
<td>Zi</td>
<td>Zi</td>
<td>Wi</td>
<td>Wi</td>
</tr>
<tr>
<td>Xh</td>
<td>Xh</td>
<td>Yh</td>
<td>Yh</td>
<td>Zh</td>
<td>Zh</td>
<td>Wh</td>
<td>Wh</td>
</tr>
<tr>
<td>Xi</td>
<td>Xi</td>
<td>Yi</td>
<td>Yi</td>
<td>Zi</td>
<td>Zi</td>
<td>Wi</td>
<td>Wi</td>
</tr>
<tr>
<td>Xh</td>
<td>Xh</td>
<td>Yh</td>
<td>Yh</td>
<td>Zh</td>
<td>Zh</td>
<td>Wh</td>
<td>Wh</td>
</tr>
<tr>
<td>Xi</td>
<td>Xi</td>
<td>Yi</td>
<td>Yi</td>
<td>Zi</td>
<td>Zi</td>
<td>Wi</td>
<td>Wi</td>
</tr>
<tr>
<td>Xh</td>
<td>Xh</td>
<td>Yh</td>
<td>Yh</td>
<td>Zh</td>
<td>Zh</td>
<td>Wh</td>
<td>Wh</td>
</tr>
</tbody>
</table>

- **c0**: DF
- **c7**: DF
### Align1 – SIMD8 - 64bit (Haswell+)

<table>
<thead>
<tr>
<th></th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B</td>
<td>Xl0</td>
<td>Xh0</td>
<td>Xl1</td>
<td>Xh1</td>
<td>Xl2</td>
<td>Xh2</td>
<td>Xl3</td>
<td>Xh3</td>
</tr>
<tr>
<td>32B</td>
<td>Xl4</td>
<td>Xh4</td>
<td>Xl5</td>
<td>Xh5</td>
<td>Xl6</td>
<td>Xh6</td>
<td>Xl7</td>
<td>Xh7</td>
</tr>
<tr>
<td>64B</td>
<td>Xl0</td>
<td>Xh0</td>
<td>Xl1</td>
<td>Xh1</td>
<td>Xl2</td>
<td>Xh2</td>
<td>Xl3</td>
<td>Xh3</td>
</tr>
<tr>
<td>96B</td>
<td>Xl4</td>
<td>Xh4</td>
<td>Xl5</td>
<td>Xh5</td>
<td>Xl6</td>
<td>Xh6</td>
<td>Xl7</td>
<td>Xh7</td>
</tr>
</tbody>
</table>

```
mov(8)  g2<1>DF  g0<4,4,1>DF  { align1 1Q }
```
Align1 – SIMD8 - 64bit (Haswell+)

```
<table>
<thead>
<tr>
<th></th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B</td>
<td>Xh0</td>
<td>Xh1</td>
<td>Xh2</td>
<td>Xh3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32B</td>
<td>Xh4</td>
<td>Xh5</td>
<td>Xh6</td>
<td>Xh7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64B</td>
<td>Xh0</td>
<td>Xh1</td>
<td>Xh2</td>
<td>Xh3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96B</td>
<td>Xh4</td>
<td>Xh5</td>
<td>Xh6</td>
<td>Xh7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Use the subscript() helper:

  ```c
  subscript(reg, BRW_REGISTER_TYPE_UD, 1)
  ```
i965 – Fp64 (Align16)
Align16 – SIMD4x2 - 32bit

```
mov(8)  g0.0<1>.xyF  g1.0<4,4,1>.ywwwF  { align16 1Q }
```

Fixed

Fixed
Align16 - 64bit

- Broadwell+ used to need Align16 for **Geometry** and **Tessellation** shaders.
  - Nowadays these platforms are **fully scalar** and Align1 support is sufficient to expose Fp64
- Older platforms (Haswell, IvyBridge, etc) still need Align16 for **Vertex**, **Geometry** and **Tessellation** shaders.
Align16 - 64bit

- **Swizzle channels are 32-bit**, even on 64-bit operands
  - We can only address DF components XY directly!
- **Writemasks are 64-bit** for DF destinations though
  - WRITEMASK_XY and WRITEMASK_ZW are 32-bit though → no native representation
Align16 - SIMD4x2 - 64bit

Vertex 1
0B  X_l  X_h  Y_l  Y_h  Z_l  Z_h  W_l  W_h  g0
Vertex 2
32B  X_l  X_h  Y_l  Y_h  Z_l  Z_h  W_l  W_h  g1
64B  
96B  

mov(8)  g2<1>.xyDF  g0<2,2,1>.zwzwDF  \{ align16 1Q \}
Align16 - SIMD4x2 - 64bit

- Align16 requires 16B alignment
  - 2 DF components in each row.
- Vstride=2 to cover the entire region

```
mov(8) \textit{g2<1>.xyDF} \quad \textit{g0<2,2,1>.zwzwDF} \quad \{ \text{align16 1Q} \}
```
Align16 - SIMD4x2 - 64bit

Each 16B region applies the 4-component 32-bit swizzle

\[
\text{mov}(8) \quad g2<1>.xyDF \quad g0<2,2,1>.zwzwDF \quad \{ \text{align16 1Q} \}
\]
Align16 - SIMD4x2 - 64bit

<table>
<thead>
<tr>
<th></th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B</td>
<td>Xi</td>
<td>Xh</td>
<td>Yi</td>
<td>Yh</td>
<td>Zi</td>
<td>Zh</td>
<td>Wi</td>
<td>Wh</td>
</tr>
<tr>
<td>32B</td>
<td>Xi</td>
<td>Xh</td>
<td>Yi</td>
<td>Yh</td>
<td>Zi</td>
<td>Zh</td>
<td>Wi</td>
<td>Wh</td>
</tr>
<tr>
<td>64B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thread boundary

mov(8)  g2<1>.xyDF  g0<2,2,1>.zwzwDF  { align16 1Q }

- We can’t do all swizzle combinations:
  - XXXX, YZYX, XYYZ, etc. are not supported.
- We need to translate our 64-bit swizzles to 32-bit.
  - X → XY, Y → ZW, Z → ?, W → ?
Align16 - SIMD4x2 - 64bit

XY / ZW component splitting

\[
\begin{array}{cccccccc}
\text{c0} & \text{c1} & \text{c2} & \text{c3} & \text{c4} & \text{c5} & \text{c6} & \text{c7} \\
0B & \text{X}_l & \text{X}_h & \text{Y}_l & \text{Y}_h & \text{X}_l & \text{X}_h & \text{Y}_l & \text{Y}_h \\
32B & \text{Z}_l & \text{Z}_h & \text{W}_l & \text{W}_h & \text{Z}_l & \text{Z}_h & \text{W}_l & \text{W}_h \\
64B & \text{Z}_l & \text{Z}_h & \text{W}_l & \text{W}_h & \text{Z}_l & \text{Z}_h & \text{W}_l & \text{W}_h \\
96B & & & \text{Y}_l & \text{Y}_h & & & \text{Y}_l & \text{Y}_h \\
\end{array}
\]

Thread boundary

- \(\text{mov}(8)\) \(\text{g2}^\langle1\rangle.\text{xywDF} \rightarrow \text{g0}^\langle2,2,1\rangle.\text{zywDF}\)
- \(\text{mov}(4)\) \(\text{g2}^\langle1\rangle.\text{xyDF} \rightarrow \text{g0}+1^\langle2,2,1\rangle.\text{xyzwDF}\)
- \(\text{mov}(4)\) \(\text{g2}+1^\langle1\rangle.\text{yDF} \rightarrow \text{g0}^\langle2,2,1\rangle.\text{zwzwDF}\)
Align16 - SIMD4x2 - 64bit
XY / ZW component splitting

<table>
<thead>
<tr>
<th></th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B</td>
<td>X_l</td>
<td>X_h</td>
<td>Y_l</td>
<td>Y_h</td>
<td>X_l</td>
<td>X_h</td>
<td>Y_l</td>
<td>Y_h</td>
</tr>
<tr>
<td>32B</td>
<td>Z_l</td>
<td>Z_h</td>
<td>W_l</td>
<td>W_h</td>
<td>Z_l</td>
<td>Z_h</td>
<td>W_l</td>
<td>W_h</td>
</tr>
<tr>
<td>64B</td>
<td>X_l</td>
<td>X_h</td>
<td></td>
<td></td>
<td>X_l</td>
<td>X_h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>96B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

emask

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Wrong!

Thread boundary

mov(4)  g2<1>.xDF  g0<2,2,1>.xyxyDF  \{ align16 1Q \}
Align16 - SIMD4x2 - 64bit

<table>
<thead>
<tr>
<th></th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B</td>
<td>Xl</td>
<td>Xh</td>
<td>Yl</td>
<td>Yh</td>
<td>Zl</td>
<td>Zh</td>
<td>Wi</td>
<td>Wh</td>
</tr>
<tr>
<td>32B</td>
<td>Xl</td>
<td>Xh</td>
<td>Yl</td>
<td>Yh</td>
<td>Zl</td>
<td>Zh</td>
<td>Wi</td>
<td>Wh</td>
</tr>
<tr>
<td>64B</td>
<td>Yl</td>
<td>Yh</td>
<td>Yl</td>
<td>Yh</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96B</td>
<td>Yl</td>
<td>Yh</td>
<td>Yl</td>
<td>Yh</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **mov(8)**  
g2<1>.xyDF  
g0<2,2,1>.zwzwDF  
{ align16 1Q }

- Back to square one…
  - Z → XY, W → ZW (at 16B offset)
### Align16 – SIMD4x2 - 64bit

<table>
<thead>
<tr>
<th>Address</th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B</td>
<td>$X_l$</td>
<td>$X_h$</td>
<td>$Y_l$</td>
<td>$Y_h$</td>
<td>$Z_l$</td>
<td>$Z_h$</td>
<td>$W_l$</td>
<td>$W_h$</td>
</tr>
<tr>
<td>32B</td>
<td>$X_l$</td>
<td>$X_h$</td>
<td>$Y_l$</td>
<td>$Y_h$</td>
<td>$Z_l$</td>
<td>$Z_h$</td>
<td>$W_l$</td>
<td>$W_h$</td>
</tr>
<tr>
<td>64B</td>
<td>$Z_l$</td>
<td>$Z_h$</td>
<td>$W_l$</td>
<td>$W_h$</td>
<td>$Z_l$</td>
<td>$Z_h$</td>
<td>$W_l$</td>
<td>$W_h$</td>
</tr>
<tr>
<td>96B</td>
<td>$Z_l$</td>
<td>$Z_h$</td>
<td>$W_l$</td>
<td>$W_h$</td>
<td>$Z_l$</td>
<td>$Z_h$</td>
<td>$W_l$</td>
<td>$W_h$</td>
</tr>
</tbody>
</table>

- **mov(8) g2<1>.xyDF g0.2<2,2,1>.xyzwDF** \{ align16 1Q \}
  - Not good enough (and violates register region restrictions)
    - We could use a combination of `vstride=0` and **SIMD splitting**.
Align16 - SIMD4x2 - 64bit

- Gen7 hardware seems to have an interesting bug feature:
  - The second half of a compressed instruction with vstride=0 will ignore the vstride and offset exactly 1 register
  - We can use this to avoid the SIMD splitting
Align16 - SIMD4x2 - 64bit

```
mov(8)  g2<1>.xyDF  g0.2<0,2,1>.xyzwDF  { align16 1Q }
```
**Align16 - SIMD4x2 - 64bit**

```
mov(8)  g2<1>.xyDF  g0.2<0,2,1>.xyzwDF  { align16 1Q }
```

- Remember that issue with 32-bit writemasks?
  - `WRITEMASK_XY == WRITEMASK_X`
### Align16 - SIMD4x2 - 64bit

<table>
<thead>
<tr>
<th></th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B</td>
<td>Xᵢ</td>
<td>Xₜ</td>
<td>Yᵢ</td>
<td>Yₜ</td>
<td>Zᵢ</td>
<td>Zₜ</td>
<td>Wᵢ</td>
<td>Wₜ</td>
</tr>
<tr>
<td>32B</td>
<td>Xᵢ</td>
<td>Xₜ</td>
<td>Yᵢ</td>
<td>Yₜ</td>
<td>Zᵢ</td>
<td>Zₜ</td>
<td>Wᵢ</td>
<td>Wₜ</td>
</tr>
<tr>
<td>64B</td>
<td>Zᵢ</td>
<td>Zₜ</td>
<td>Wᵢ</td>
<td>Wₜ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96B</td>
<td>Zᵢ</td>
<td>Zₜ</td>
<td>Wᵢ</td>
<td>Wₜ</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**mov(8)**
- g2<1>.xDF
- g2<1>.yDF

**g0.2<0,2,1>.xyzwDF**

{ align16 1Q }

{ align16 1Q }
Align16 - SIMD4x2 - 64bit

• This is just an example:
  • Different swizzle combinations may require different implementations
  • 2-src instructions and 3-src instructions
Align16 - SIMD4x2 - 64bit

- Implementation:
  - **Step 1**: scalarize everything, *swizzle translation at codegen* - Done
  - **Step 2**: let through swizzle classes that we can support natively (e.g. XYZW) - Done
  - **Step 3**: let through swizzle classes that we can support by exploiting the vstride=0 behavior (e.g. XXXX) - Done
  - **Step 4**: use component splitting (partial scalarization) to support more swizzle classes – Not Done (yet)
i965 – Fp64
Common Issues
Multiple hardware generations

- Significant differences between IvyBridge, Haswell and Broadwell+ hardware

- Skylake did not require specific adaptations
  - Broxton, CherryView and Braswell only required minor tweaks:
    - 32b to 64b conversions need 64b aligned source data
    - 64b indirect addressing not supported
32-bit driver

- Before fp64 all GLSL types were implemented as 32-bit types.
  - Driver code assumed 32-bit types (and even hstride=1) in lots of places.
- Many fixes like:
  - \( \text{int dst\_width} = \text{inst->exec\_size} / 8; \)
  + \( \text{int dst\_width} = \text{DIV\_ROUND\_UP}(\text{inst->dst.component\_size(\text{inst->exec\_size}), REG\_SIZE}); \)
- This could happen anywhere in the driver
  - Piglit was the driving force to find these
Unfamiliar code patterns

- Fp64 operation produces new code patterns:
  - 32-bit access patterns on low/high 32-bit chunks of 64-bit data
  - Horizontal strides $\neq 1$
- Some parts of the driver did not handle these scenarios properly.
  - Copy-propagation received at least 7 patches!
32-bit read/write messages

- All read/write messages are 32-bit
  - Pull loads, UBOs, SSBOs, URB, scratch...
- 64bit data needs to be shuffled into 32-bit channels before writing
- 32bit data reads need to be shuffled into valid 64bit data channels
  - shuffle_64bit_data_for_32bit_write()
  - shuffle_32bit_load_result_to_64bit_data()
32-bit read/write messages (SIMD8 read)

- Just what we want for 32-bit scalar operation
  - 8x16B SIMD8 read messages
  - Separate variables (registers) for each component
  - Consecutive components in consecutive registers
### 32-bit read/write messages (SIMD8 read)

<table>
<thead>
<tr>
<th></th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B</td>
<td>Xi</td>
<td>Xi</td>
<td>Xi</td>
<td>Xi</td>
<td>Xi</td>
<td>Xi</td>
<td>Xi</td>
<td>Xi</td>
</tr>
<tr>
<td>32B</td>
<td>Xh</td>
<td>Xh</td>
<td>Xh</td>
<td>Xh</td>
<td>Xh</td>
<td>Xh</td>
<td>Xh</td>
<td>Xh</td>
</tr>
<tr>
<td>64B</td>
<td>Yi</td>
<td>Yi</td>
<td>Yi</td>
<td>Yi</td>
<td>Yi</td>
<td>Yi</td>
<td>Yi</td>
<td>Yi</td>
</tr>
<tr>
<td>96B</td>
<td>Yh</td>
<td>Yh</td>
<td>Yh</td>
<td>Yh</td>
<td>Yh</td>
<td>Yh</td>
<td>Yh</td>
<td>Yh</td>
</tr>
</tbody>
</table>

- **read(+0B)**
- **Invalid 64b data**

<table>
<thead>
<tr>
<th></th>
<th>g0</th>
<th>g1</th>
<th>g2</th>
<th>g3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xi</td>
<td>Xi</td>
<td>Xi</td>
<td>Xi</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
</tr>
</thead>
<tbody>
<tr>
<td>128B</td>
<td>Zi</td>
<td>Zi</td>
<td>Zi</td>
<td>Zi</td>
<td>Zi</td>
<td>Zi</td>
<td>Zi</td>
<td>Zi</td>
</tr>
<tr>
<td>160B</td>
<td>Zh</td>
<td>Zh</td>
<td>Zh</td>
<td>Zh</td>
<td>Zh</td>
<td>Zh</td>
<td>Zh</td>
<td>Zh</td>
</tr>
<tr>
<td>192B</td>
<td>Wi</td>
<td>Wi</td>
<td>Wi</td>
<td>Wi</td>
<td>Wi</td>
<td>Wi</td>
<td>Wi</td>
<td>Wi</td>
</tr>
<tr>
<td>224B</td>
<td>Wh</td>
<td>Wh</td>
<td>Wh</td>
<td>Wh</td>
<td>Wh</td>
<td>Wh</td>
<td>Wh</td>
<td>Wh</td>
</tr>
</tbody>
</table>
## 32-bit read/write messages (SIMD8 read)

<table>
<thead>
<tr>
<th></th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B</td>
<td>Xi</td>
<td>Xh</td>
<td>Xi</td>
<td>Xh</td>
<td>Xi</td>
<td>Xh</td>
<td>Xi</td>
<td>Xh</td>
</tr>
<tr>
<td>32B</td>
<td>Xi</td>
<td>Xh</td>
<td>Xi</td>
<td>Xh</td>
<td>Xi</td>
<td>Xh</td>
<td>Xi</td>
<td>Xh</td>
</tr>
<tr>
<td>64B</td>
<td>Yi</td>
<td>Yh</td>
<td>Yi</td>
<td>Yh</td>
<td>Yi</td>
<td>Yh</td>
<td>Yi</td>
<td>Yh</td>
</tr>
<tr>
<td>96B</td>
<td>Yi</td>
<td>Yh</td>
<td>Yi</td>
<td>Yh</td>
<td>Yi</td>
<td>Yh</td>
<td>Yi</td>
<td>Yh</td>
</tr>
</tbody>
</table>

For read(+0B):

- g0: Xi, Xh, Xi, Xh, Xi, Xh, Xi, Xh
- g1: Xi, Xh, Xi, Xh, Xi, Xh, Xi, Xh
- g2: Yi, Yh, Yi, Yh, Yi, Yh, Yi, Yh
- g3: Yi, Yh, Yi, Yh, Yi, Yh, Yi, Yh

For read(+16B):

- g4: Zi, Zh, Zi, Zh, Zi, Zh, Zi, Zh
- g5: Zi, Zh, Zi, Zh, Zi, Zh, Zi, Zh
- g6: Wi, Wh, Wi, Wh, Wi, Wh, Wi, Wh
- g7: Wi, Wh, Wi, Wh, Wi, Wh, Wi, Wh
64-bit immediates (gen7)

- **No support for 64-bit immediates** *(Haswell, IvyBridge)*
  - Haswell provides the *DIM* instruction specifically for this purpose, we just had to add support for this in the driver.
  - IvyBridge requires that we emit code to load each 32-bit chunk of the constant into a register and then return either a XXXX swizzle (align16) or a stride 0 (align1).
Bugs / restrictions (Align1)

- Second half of compressed instructions that don’t write all channels have wrong emask (Haswell)
  - Requires SIMD splitting
- Second half of compressed 64-bit instructions has wrong emask (IvyBridge)
  - Requires unconditional SIMD splitting of all 64-bit instructions :-(

Bugs / restrictions (Align16)

- **Vertical stride 0 doesn’t work** *(gen7)*
  - The second half of a compressed instruction will invariably offset a full register
  - Requires SIMD splitting

- **Instructions that write 2 registers must also read 2 registers** *(gen7)*
  - This was a known issue in gen7, but it was never triggered in Align16 before Fp64
  - Requires SIMD splitting

A simple test that just copies a DF uniform to the output hits both of these bugs! :-(
Bugs / restrictions (Align16)

- 3-src instructions can’t use RepCtrl=1
  - Not supported for 64-bit instructions
  - Only affects to MAD
  - RepCtrl=0 leads to <4,4,1>:DF regions so it can only be used to work with components XY
    - Requires temporaries and component splitting, but leads to quite bad code in general
    - Avoiding MAD altogether seems a better option for now
Bugs / restrictions (Align16)

- **Compressed bcSEL**
  - Does not read the predication mask properly
  - Requires SIMD splitting
- **Dependency control**
  - Can’t be used with 64-bit instructions → GPU hangs
Current State
Status

- **Skylake**: Available in Mesa 12.0
- **Broadwell**: Available in Mesa 12.0
- **Haswell**:
  - ARB_gpu_shader_fp64: Implemented, in review
  - ARB_vertex_attrib_64bit: Implemented
- **IvyBridge**:
  - ARB_gpu_shader_fp64: Align1 implemented, Align16 implementation in progress
  - ARB_vertex_attrib_64bit: Not started
Questions?