AMD
Smarter Choice

AMD IL

AMD IL

Last time we introduced HLSL and the R600 ISA

AMD IL is a portable *immediate language* that sits between high level languages (Brook+ or HLSL) and the ISA

AMD IL is meant to be generation compatible s.t. future hardware can compile from IL whereas the ISA is asic dependent

Example

Brook+ Kernel

```
kernel void sum(float a<>, float b<>, out float c<>)
{
    c = a + b;
}
```

Generated AMD IL

```
il_ps_2_0
dcl_cb cb0[1]
dcl_resource_id(0)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)
dcl_input_generic_interp(linear) v0.xy__
dcl_resource_id(1)_type(2d,unnorm)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)
dcl_input_generic_interp(linear) v1.xy__
sample_resource(0)_sampler(0) r0.x, v0.xy00
sample_resource(1)_sampler(1) r1.x, v1.xy00
mov r2.x, r0.xxxx
mov r3.x, r1.xxxx
call 0
mov r4.x, r5.xxxx
dcl_output_generic o0
mov o0, r4.xxxx
ret
func 0
add r6.x, r2.xxxx, r3.xxxx
mov r7.x, r6.xxxx
mov r5.x, r7.xxxx
ret
end
```

IL code generation

DX HLSL

- Compile to DX asm using fxc (Microsoft HLSL compiler)
- Compile DX asm to IL using AMD GPU Shader Analyzer

AMD HLSL

- Compile AMD HLSL to IL using AMD HLSL compiler

Brook+

- Compile Brook+ kernels to IL using brcc

Or write it yourself

Writing IL code

IL resembles DX assembly

IL is also used for DirectX and OpenGL shaders

IL code will be optimized by the GPU compiler to ISA

Readability may be more important when writing IL

DX asm vs IL

```
ps_4_0
dcl_input linear v0.xy
dcl_output o0.xyzw
dcl_sampler s0, mode_default
dcl_sampler s1, mode_default
dcl_resource_texture2d ( float , float , float , float ) t0
dcl_resource_texture2d ( float , float , float , float ) t1
dcl_temps 2
sample r0.xyzw, v0.xyxx, t0.xyzw, s0
sample r1.xyzw, v0.xyxx, t1.xyzw, s1
add o0.xyzw, r0.xyzw, r1.xyzw
ret
```

DX asm

```
il_ps_2_0
dcl_input_interp(linear) v0.xy__
dcl_output_generic o0
dcl_resource_id(0)_type(2d)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)
dcl_resource_id(1)_type(2d)_fmtx(float)_fmty(float)_fmtz(float)_fmtw(float)
sample_resource(0)_sampler(0) r0, v0.xyxx
sample_resource(1)_sampler(1) r1, v0.xyxx
add o0, r0, r1
ret_dyn
end
```

AMD IL

Instruction syntax

<instr>[_<ctrl>][_<ctrl(val)>] [<dst>[_<mod>][.<write-mask>]] [, <src>[_<mod>][.<swizzle-mask>]]...

Broken down:

<instr> [_<ctrl>][_<ctrl(val)>]
- –instruction with control specifiers

[<dst>[_<mod>][.<write-mask>]]
- –destination register with modifier and write mask

[, <src>[_<mod>][.<swizzle-mask>]]...
- –source registers with modifier and swizzle mask

Registers

Registers are four component vectors

v# - import registers

o# - output registers

r# - general purpose registers

There are also other special enumerated registers

Registers are typeless. Integer instructions can operate on float data. User must take care to keep track of register types and convert

Register modifiers

Destination modifiers apply extra operations to the destination register after the instruction runs

Examples:

- <dst>_x2 multiplies by 2
- <dst>_d4 divides by 4

Source modifiers apply extra operations to the source register before the instruction runs

Examples:

- <src>_abs returns the absolute value
- <src>_sign returns the sign

Write Masks

Element-wise write masks on destination registers control how components are written to

Syntax: reg.{x|_|0|1}{y|_|0|1}{z|_|0|1}{w|_|0|1}

- x,y,z,w are components
- underscore "_", means don't write (can also leave blank)
- 0 or 1, replaces component with 0 or 1

Mask is position dependent

Examples:

- mov r0.x___, r1; move r1.x to r0.x and leave rest unchanged
- mov r0.x, r1; same as previous
- mov r0.y_w, r1; only write to r0.y and r0.w and leave rest
- mov r0.0000, r1; zero out all components
- mov r0.xyz1, r1; write all except w, which changes to 1

Swizzle mask

Controls how the register components are used

Syntax:

reg.{x|y|z|w|0|1}{x|y|z|w|0|1}{x|y|z|w|0|1}{x|y|z|w|0|1}

Mask is position independent

Blanks mean use default component

Examples:

- mov r0, r1.yxzw; move r1.y->r0.x, r1.x->r0.y, r1.z->r0.z, r1.w->r0.w
- mov r0, r1.yx; same as previous
- mov r0, r1.xyz0; standard move except force r0.w to zero

Instructions

Declaration and Initialization

Input (memory fetches)

Conversion

General ALU instructions (Math/Trig/Special)

Flow control

Bitwise

Double

Comparison

Declaration and initialization

Inputs and outputs must be declared
- constant buffers
- input interpolators
- resources (textures/streams)
- literals

Constant buffers

In the past, constants were passed to the shader individually

Now, constants are passed together in constant buffers

Constant buffers elements are 4 component vectors

Declaration:

dcl_cb cb<#>[<*size*>]

Buffer elements are addressed like C arrays

Constant literals

Literals are constant values used in the source code. For example, int a = 5;

Previously, constants had to be passed in like constant variables

Syntax:

dcl_literal /#, <x-bits>, <y-bits>, <z-bits>, <w-bits>

Example:

dcl_literal l1, 0x40A00000, 0x3F800000, 0x3E99999A, 0x3E0F5C29
  –float literal float4(5, 1, 0.3, 0.14)

dcl_literal l2, 0x00000003, 0xFFFFFFFF, 0xFFFFFFFB, 0x00000007
  –integer literal int4(3, -1, -5, 7)

Declaring memory

Syntax:

dcl_resource_id(*n*)_type(*pixtexusage*[,unorm])_fmtx(*fmt*)
_fmty(*fmt*)_fmtz(*fmt*)_fmtw(*fmt*)

Example:

dcl_resource_id(0)_type(2d)_fmtx(float)_fmty(float)_fmtz
(float)_fmtw(float)

- –2D texture of float4

Fetching from memory/textures

Textures can be addressed as normalized (0...1) or unormalized. It depends on how it was declared (unorm)

Syntax:

sample_resource(*n*)_sampler(*m*)[_aoffimmi(*u*, *v*, *w*)] *dst*, *src0*

aoffimmi apply integer offset to the input address

Addressing is determined by the type of the texture (1D vs. 2D)

Scratch buffers

You can allocate a temporary buffer that you can address like C arrays called scratch buffers

Performance can be slow, but you can address using registers

Example:

dcl_index_temp_array x0[*size*]

mov x0[r0.x], r3

mov r2, x0[r0.y]

General ALU instructions

See the spec for a complete list of functions

There are often different integer, double, and float instructions (e.g., IADD, DADD, ADD respectively)

There are also special functions (e.g., CMOV, FLR)

There are also trig instructions (e.g., SIN, COS)

Doubles

Doubles are a special case because there are no native 64-bit component registers

Doubles are represented as two 32-bit components

If *src*.xy contains 0x4008000000000000, then *src*.y = 0x40080000 and *src*.x = 0x00000000

When using double, the value must be in positions *src*.xy

See spec for the double operators

Conversion

Registers are typeless, so registers need to be converted from one type to another

Be careful, conversions only happen on the transcendental unit

Valid conversions are:

- D2F
- F2D
- FTOI
- FTOU
- ITOF
- UTOF

Flow control instructions

Basic branching

if_logicalnz *src0*.x //execute instruction block if *scr0*.x!=0

if_logicalz *src0*.x //execute instruction block if *src0*.x==0

ifc_relop(*op*) *src0*, *src1* //execute block if relop is true
  –*op*: eq (==), ne (!=), gt (>), ge (>=), lt (<), le (<=)

else

endif

Control flow instructions (cont.)

Loops

whileloop

break instruction

- –break
- –break_logicalnz *src0*
- –break_logicalz *src0*
- –breakc_relop(*op*) *src0*, *src1*

endloop

Comparison

You can break on a register by first doing a comparison instruction first

- –eq, ge, lt, ne

Output

Kernel outputs are written to special output registers

They can only be used for output, but are handled like any other register up to the maximum number

Global gather/scatter buffers

Scatter is handled using a global buffer

There is only one global buffer at any time, so it is not declared

It is addressed like scratch buffers using a C array interface

Because it uses 128 bit alignment, it is always treated as a float4 type, so be careful when addressing

Example:

```
mov g[r0.x], r0

mov r1, g[r3.x]
```

Lots more…

See the spec for more info

Learn by example:

- Look at the CAL samples
- Use AMD's GPU ShaderAnalyzer to generate IL from HLSL or Brook+