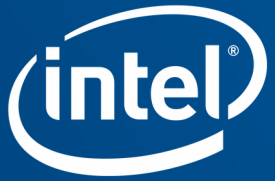


An Introduction to i965 Assembly and Bit Twiddling Hacks

Matt Turner – X.Org Developer’s Conference 2018

Objectives

- Introduce i965 instruction assembly
  - At least enough to know what you're looking at
- Tell you how it's different from other GPUs
- Demonstrate some interesting optimizations it allows
- Show our method of verifying instructions are valid

Assumptions

- Probably already familiar with some assembly language
- If you’re here, maybe familiar with a GPU assembly language
- Probably know of weird architectures or instructions
  - Maybe know CPUs because of weird instructions

Intel Gen Graphics (i965)

- “i965” is the name of Intel’s graphics core from 2006
- We call that Gen4 graphics
- Everything since then is a descendant
  - E.g., Ironlake, Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake, Kaby Lake, …
- Instruction set changes like the rest of the hardware with each generation
  - But still very recognizable

i965 instruction set features

In common with other GPUs

- Source and destination modifiers
  - source: neg, abs, neg+abs; dest: saturate
- Instruction predication
  - Ability to nullify an instruction
- Unified register file
  - Integer and floating-point use same registers

Less common features

- Conditional modifiers
- Mixed type operations
  - Fewer each generation
- Vector immediate values
- Register regioning

Common features

- Unified register file
  - Can operate on floating-point data as integer in same register (and vice versa)
  - 128 256-bit registers, usable as 8x floats, 4x doubles, 16x words, etc.
- Source modifiers
  - Written as “-”, “(abs)”, “-(abs)” (and sometimes “~”) before a source operand
- Saturate (clamp result to 0.0 to 1.0)
  - Written as “.sat” suffix on instruction mnemonic
- Instruction predication
  - Written as “(condition)” before instruction, uses a special flag register

Trivial i965 program (glxgears fragment shader)

```
pln(8)   g124<1>F   g4<0,1,0>F    g2<8,8,1>F
pln(8)   g125<1>F   g4.4<0,1,0>F  g2<8,8,1>F
pln(8)   g126<1>F   g5<0,1,0>F    g2<8,8,1>F
pln(8)   g127<1>F   g5.4<0,1,0>F  g2<8,8,1>F
sendc(8) null<1>UW  g124<8,8,1>F
   render RT write SIMD8 LastRT Surface = 0 mlen 4 rlen 0
```

i965 instruction set is different (but familiar...)

- GPU instruction sets are necessarily different than CPU ISAs
- Designed to execute massively parallel programs
- Today most GPU ISAs appear scalar (SPMD model)
  - Compilers are good at scalar code
  - Compiler doesn’t need to know how big that “vector register” is
- i965 looks like AVX2 with channel masking (SIMD model)
  - Exposes vector architecture to compiler writer
  - Compiler must consider cross-channel interference
  - But offers lots of flexibility

Breaking it down

op(exec size) dest<stride>type src0<stride>type src1<stride>type

- op – opcode. E.g., add, mul, mov, sel, send, etc.
- execution size – Number of channels to operate on
- dest, src0, src1 – Operands
  - Includes register file, register number, subregister number
- stride – Parameters describing order registers' channels will be read
- type – Operand data type
  - Common types: F (float), D (32-bit doubleword), UD (32-bit unsigned)

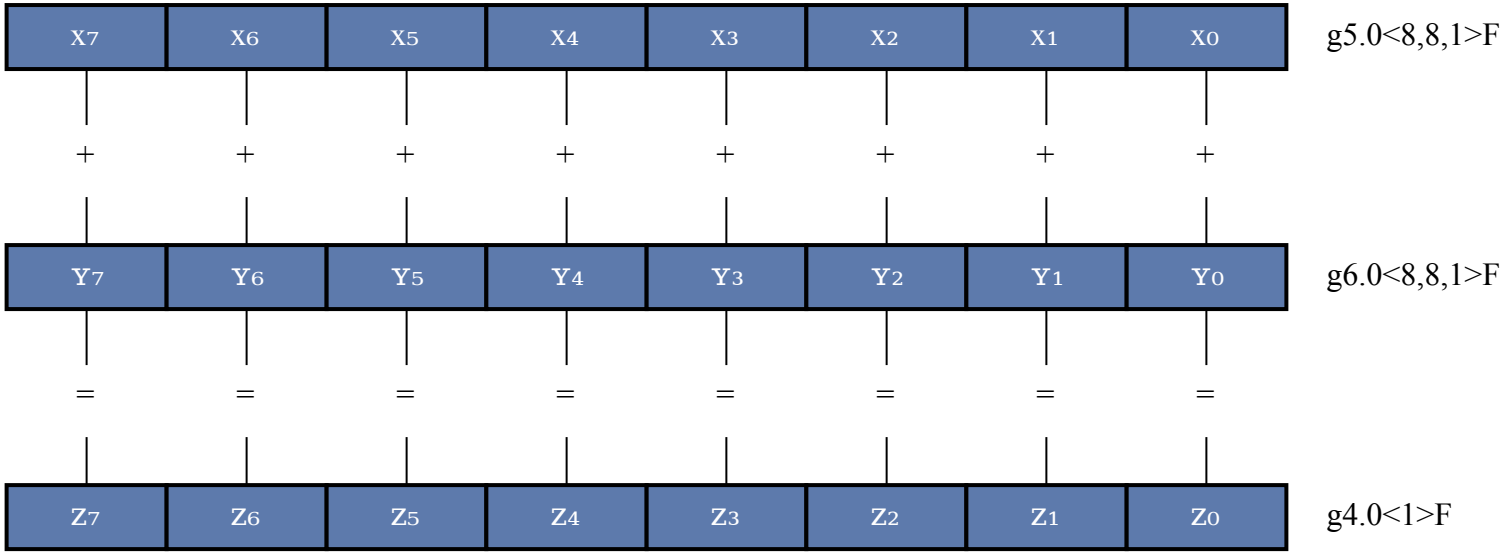
Basic floating-point addition

op(exec size)	dest<stride>type	src0<stride>type	src1<stride>type
add(8)	g4<1>F	g5<8,8,1>F	g6<8,8,1>F

- Adds 8 (exec size)
  - Consecutive floats in general register #5 with
  - Consecutive floats in general register #6
  - Storing in consecutive float channels of general register #4

Basic floating-point addition

add(8) g4<1>F g5<8,8,1>F g6<8,8,1>F

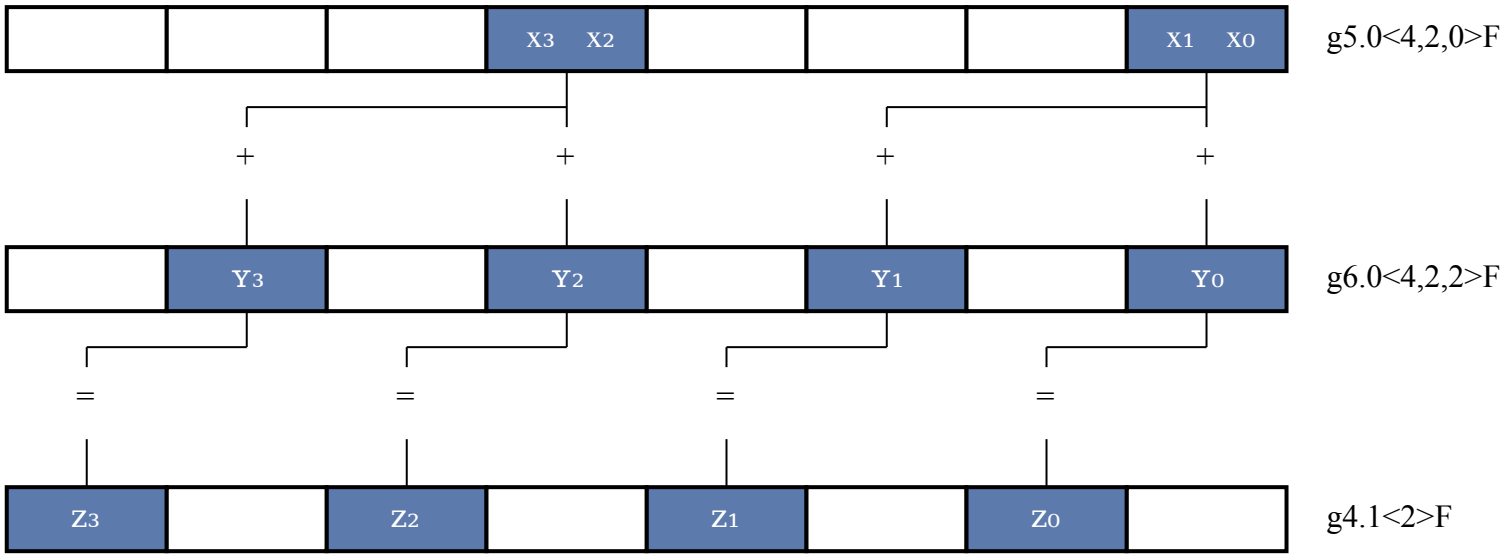


Register Regioning

- Parameters of the <stride> define a *register region*
  - Defines the manner in which the registers channels are accessed
- Destination has a single parameter (just called stride) that skips components
- Sources have three parameters
  - Vertical stride, width, horizontal stride, written <V,W,H>

Register Regioning example

add(4) g4.1<2>F g5<4,2,0>F g6<4,2,2>F



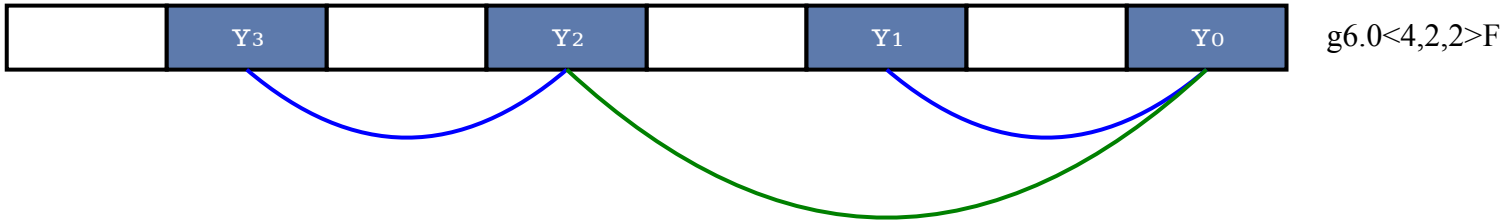
Source Register Regioning

- Best interpreted by reading them backwards
  - Striding *horizontally, accessing width* channels
  - Then stride *vertically* from the beginning of the “width”
  - Repeat striding horizontally, then vertically until *exec size* channels have been accessed

Register Regioning example

add(4) g4.1<2>F g5<4,2,0>F g6<4,2,2>F

- Access 2 (width) channels by striding by 2 (horizontally)
- Then stride by 4 (vertically)



Register Regioning key points

- Only a few register regions are common
  - <8,8,1> - standard “read the channels in order”
  - <0,1,0> - uniform “read the same channel” *exec size* times
  - <0,4,1> - vec4 uniform “read same four channels in order”
- Equivalent regions can be described in multiple ways
- **Many restrictions on what combinations are legal**
  - Must consider all operand regions, subregister, etc, to determine legality
  - Difficult for a human to quickly determine whether an instruction is legal

bool to float

mov(8) g3<1>F -g2<8,8,1>D

- Integer True represented by all-ones (-1) and False represented by 0
- Want float 1.0f for true and 0.0f for false
- Implement with a type-converting move and a negation modifier

gl_FrontFacing

- GLSL built-in variable that indicates if primitive is front or backfacing
- *Thread payload* contains backfacing bit in bit 15

```
not(8)          g2<1>D          g0<0,1,0>D
shl(8)          g3<1>D          g2<8,8,1>D      16D
asr(8)          g4<1>D          g3<8,8,1>D      31D
```

gl_FrontFacing

- GLSL built-in variable that indicates if primitive is front or backfacing
- *Thread payload* contains backfacing bit in bit 15

```
not(8)          g2<1>D          g0<0,1,0>D
asr(8)          g4<1>D          g2<8,4,2>W      15D
```

gl_FrontFacing, a realization

- Backfacing bit is the high bit — the sign bit — of a 16-bit word
- Could use negation source modifier to flip that bit… except for 0
- Low bits of payload are primitive topology, and it must be non-zero!

gl_FrontFacing, a realization

asr(8) g2<1>D -g0<0,1,0>W 15D

- Backfacing bit is the high bit — the sign bit — of a 16-bit word
- Could use negation source modifier to flip that bit… except for 0
- Low bits of payload are primitive topology, and it must be non-zero!
- All in one instruction
  - Negate to flip high bit
  - Arithmetic shift right to fill low 16 bits
  - Sign-extend result to fill high 16-bits

sign(float x)

- Returns 1.0 if x > 0.0; -1.0 for x < 0.0; 0.0 for x == 0.0

```
cmp.g.f0(8)      null<1>F          g2<0,1,0>F       0F
(+f0) if(8)
  mov(8)         g127<1>F          1.0F
else(8)
  cmp.l.f0(8)    g3<1>F            g2<0,1,0>F       0F
  mov(8)         g4<1>F            -g3<8,8,1>D
  mov(8)         g127<1>F          -g4<8,8,1>F
endif(8)
```

sign(float x), better

- Operate on float's bits directly
  - Extract sign bit
  - Conditionally OR in 1.0f (0x3f800000) if input is non-zero

```
cmp.nz.f0(8)    null          g2<8,8,1>F     0F
and(8)          g3<1>UD       g2<8,8,1>UD    0x80000000UD
(+f0) or(8)     g3<1>UD       g3<8,8,1>UD    0x3f800000UD
```

sign(float x), better

- Operate on float's bits directly
  - Extract sign bit
  - Conditionally OR in 1.0f (0x3f800000) if input is non-zero

```
add(8)          g2<1>F          g10<8,8,1>F     g11<8,8,1>F
cmp.nz.f0(8)    null            g2<8,8,1>F      0F
and(8)          g3<1>UD         g2<8,8,1>UD     0x80000000UD
(+f0) or(8)     g3<1>UD         g3<8,8,1>UD     0x3f800000UD
```

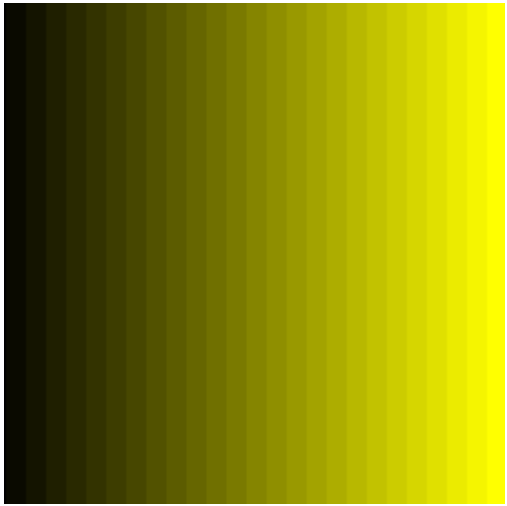
sign(float x), better

- Operate on float's bits directly
  - Extract sign bit
  - Conditionally OR in 1.0f (0x3f800000) if input is non-zero

```
add.nz.f0(8)    g2<1>F          g10<8,8,1>F     g11<8,8,1>F

and(8)          g3<1>UD         g2<8,8,1>UD     0x80000000UD
(+f0) or(8)     g3<1>UD         g3<8,8,1>UD     0x3f800000UD
```

tests/shaders/glsl-fs-integer-multiplication



```
add(16)    g2<1>UW      g1.4<1,4,0>UW    0x11001010V
mov(8)     g126<1>F     [0F 0F,0F,0F]VF
mov(8)     g127<1>F     1F
mov(8)     g3<1>F       g2<8,4,1>UW
add(8)     g5<1>F       g3<8,8,1>F       0.5F
mov(8)     g6<1>D       g5<8,8,1>F
mul(8)     g7<1>D       g6<8,8,1>D       g4<0,1,0>D
mov(8)     g8<1>F       g7<8,8,1>D
mul(8)     g9<1>F       g8<8,8,1>F       0.01F
rndd(8)    g10<1>F      -g9<8,8,1>F
mul(8)     g125<1>F     -g10<8,8,1>F     0.04F
mov(8)     g124<1>F     g125<8,8,1>F
```

Complexity even in simple cases

- At least 10 different architectural features in use
- Lots of knobs, even more restrictions
  - On regioning (very complex)
  - On source mods, operand types, saturate, conditional-mod, per-instruction
  - Restrictions change each generation
- Not simple to inspect a program and verify restrictions are not violated
  - I feel this way after six years of practice
  - How can I expect those less experienced to do this?

Validate the generated assembly

mesa/src/intel/compiler/brw_eu_validate.c

- Validates 8 classes of problems
  - Around 50 restrictions checked in total
  - Includes all register regioning restrictions (which are the easiest to miss)
- Nearly exhaustive unit testing
- Automatically validates generated shader programs in debug builds
- Optionally validates with INTEL_DEBUG={fs,vs,cs,…} envar

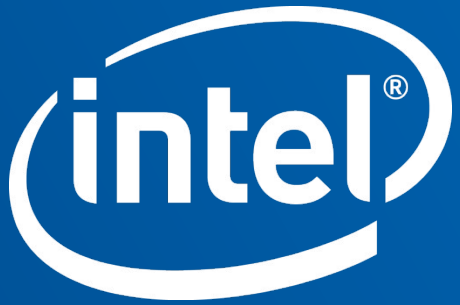
```
add(8)          g0<1>F          g0<8,8,1>F      g0<8,16,1>F
        ERROR: ExecSize must be greater than or equal to Width
```

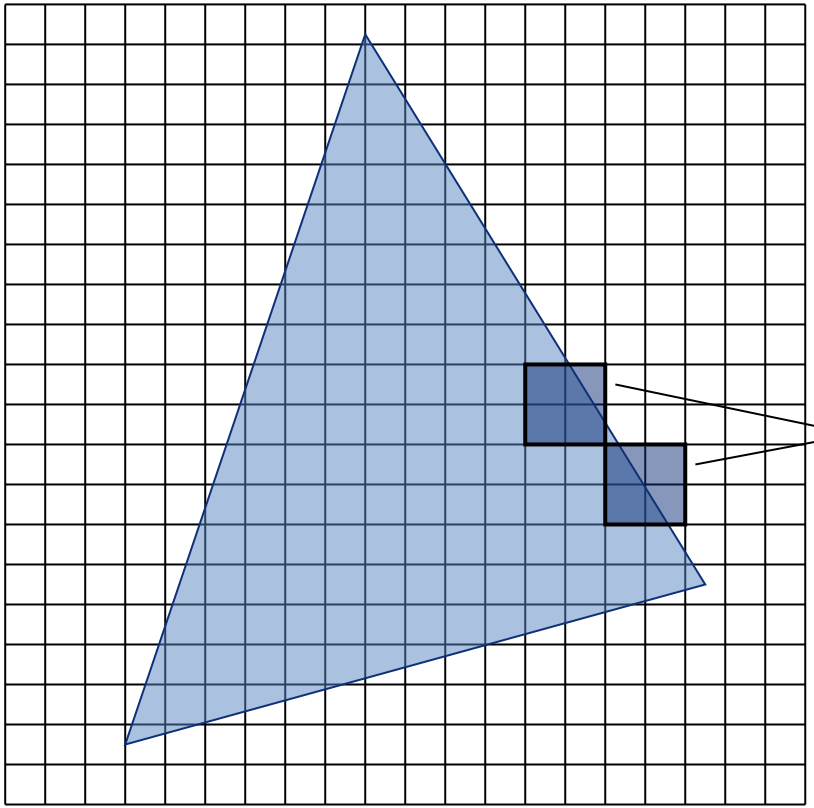
Post-mortem debugging

- Things still slip through
  - Not all restrictions are checked (yet)
  - Validator doesn’t run in release builds
- Kernel v4.13 captures compiled shaders in error state
- aubinator_error_decode runs validator on error states
  - Improved validator capable of detecting previously undetected problems

i965 instruction set is complex

- But manageably so with some guard rails
- Offers interesting optimization possibilities
  - More than just bit-twiddling hacks
- Challenging and rewarding to apply knowledge of i965 instruction set to optimize apps
- I hope this talk enables you to do just that!


intel®


Two 2x2 subspans
(a SIMD8 fragment
shader invocation)

gl_HelperInvocation

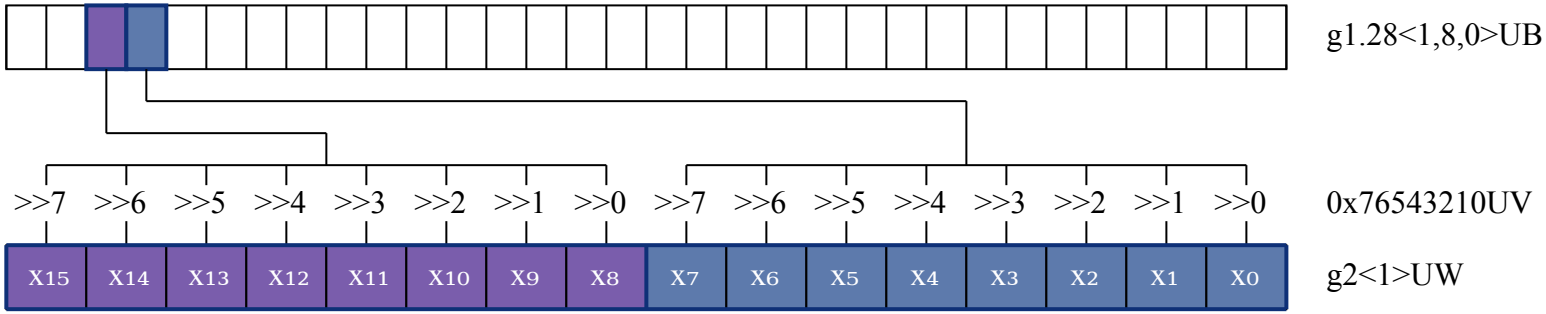
- Indicates whether an invocation is a *helper*
  - Only used for calculating derivatives, etc.
- Information provided in *thread payload* as a pixel mask
  - Again opposite of what we need; Set bit if not a helper

```
mov(1)          f0<1>UW         g1.14<0,1,0>UW
mov(8)          g2<1>D          0D
(+f0) sel(8)    g3<1>D          g2<8,8,1>D      -1D
```

gl_HelperInvocation

shr(8) g2<1>UW g1.28<1,8,0>UB 0x76543210UV

- Right shift with vector immediate
  - Gets bit into the right location
  - Garbage in high bits, and bit is still opposite of what we need



gl_HelperInvocation

and(8) g3<1>UD ~g2<8,8,1>UW 0x0001UW

- Need to clean up shift's result
  - Garbage in high bits
  - Low bit is still opposite of what we need
- Negate source modifier on and/or/xor on Broadwell+ performs bitwise-not
- Gives us 0/1
  - Now just a negation (likely free!) converts to canonical true/false representations
- Two instructions, no flag register used