Compiling to a VLIW fragment pipeline

Bill Mark and Kekoa Proudfoot

Stanford University

http://graphics.stanford.edu/projects/shading/

Goal: Movie-quality graphics in real time

Toy Story
Image Courtesy of Disney

The opportunity

Current generation of hardware is very capable

- Register-machine vertex hardware
- Multiple textures per pass
- Register-machine fragment hardware

The problem

Hardware is difficult to program

- Programming is like writing microcode
- Hard to coordinate host,
  vertex, and fragment code
- Must rewrite code for each HW platform

```
SGE R1.x, v[0].z, c[17].y;
MAD R1.z, R1.x, -c[17].y, c[17].y;
MAD R1.z, R1.x, c[18].w, R1.z ;
SLT R1.y, c[19].w, v[4].x;
MAD R1.x, R1.y, -c[18].w, c[18].w;
```

Real-time shading languages

Previous systems

- PixelFlow [Olano98]
- Single instruction per pass [Peercy00]
- Quake III [Jaquays99]

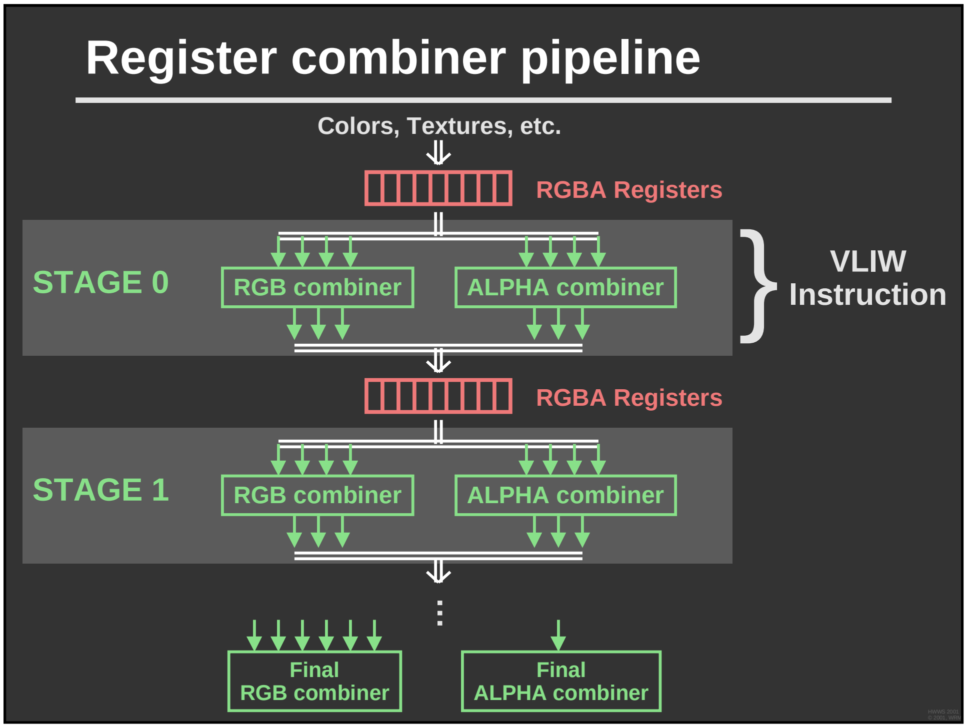
Stanford real-time shading system

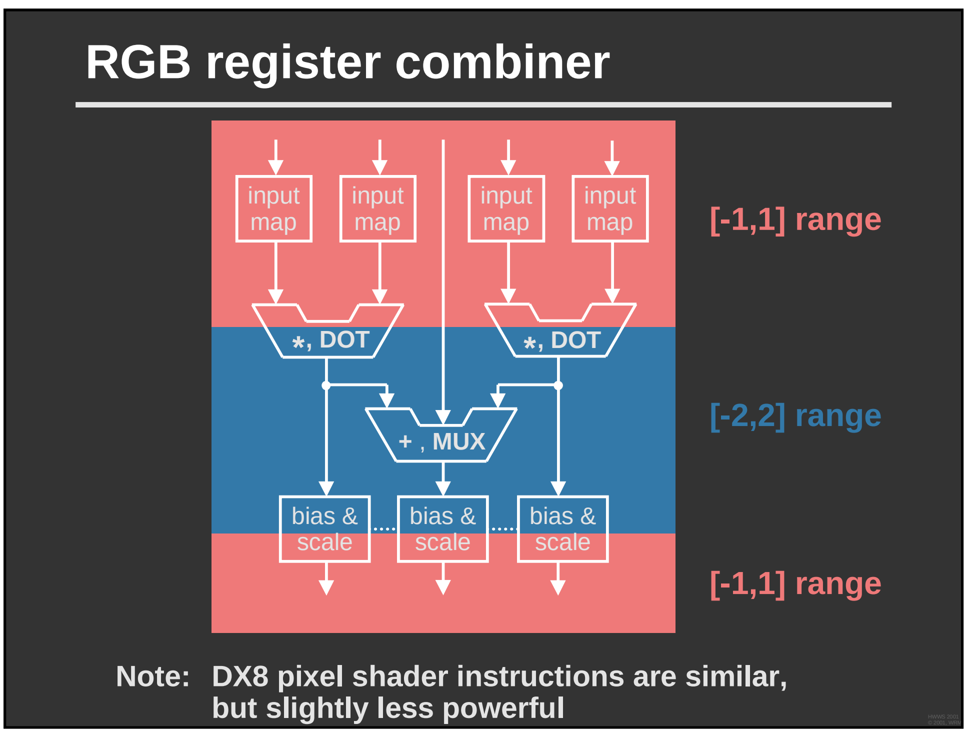
- Express fragment, vertex, and primitive-group operations in a single language
- Programmable pipeline abstraction
- Modular compiler back ends

This talk

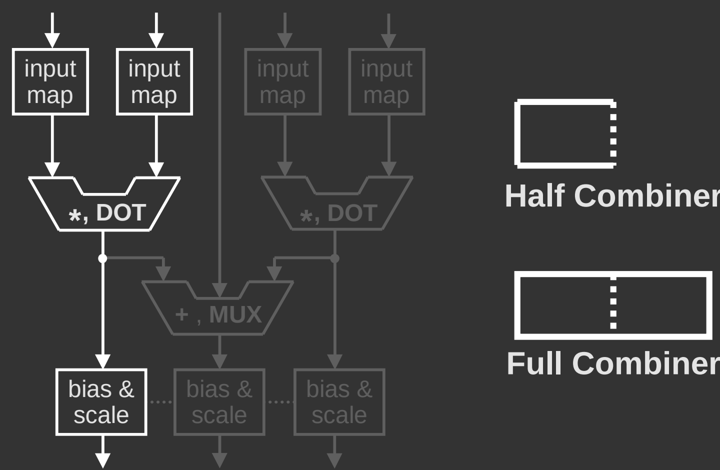
Compiler back end for register combiners

- One of three fragment backends in our system
- Targets GeForce1, 2, and 3
- Most complex back end in our system
  - Critical for performance: lots of fragments
- Supports texture shaders too
- No multi-pass yet


Register combiner pipeline
Colors, Textures, etc.
RGBA Registers
STAGE 0
RGB combiner
ALPHA combiner
VLIW Instruction
RGBA Registers
STAGE 1
RGB combiner
ALPHA combiner
Final RGB combiner
Final ALPHA combiner


RGB register combiner
input map
input map
input map
input map
[-1,1] range
*, DOT
*, DOT
+ , MUX
[-2,2] range
bias & scale
bias & scale
bias & scale
[-1,1] range
Note: DX8 pixel shader instructions are similar, but slightly less powerful

“Half” RGB combiner



An example shader

```
surface shader float4
bowling_pin (texref pinbasemarks, texref pindecals,
                    texref marksbumpF, float4 uv) {
  // Vertex code omitted
  float4 Decals = texture(pindecals, uv_decals);
  float4 basemarks = texture(pinbasemarks, uv_basemarks);
  float   Marks = alpha(basemarks);
  float3 Base  = rgb(basemarks);
  float3 Ma = {.4,.4,.4};
  float3 Md = {.5,.5,.5};
  float3 Ms = {.3,.3,.3};
  float3 Kd = rgb((Decals over {Base, 1.0}) * Marks);
  float3 C = lightmodel_bumps(Kd * Ma, Kd * Md, Ms,
                                            marksbumpF, uv_basemarks);
  return {C, 1.0};
} // bowling_pin
```

Part of bump-mapping routine

```
...

// Specular
perlight float3 Hlookup = cubenorm(Htan);
perlight float3 Hnorm   = 2.0*(Hlookup-{.5,.5,.5});
perlight float   NdotH    = clamp01(dot(Nbump, Hnorm));
perlight float   NdotHs  = select(Hlookup[2] >= 0.5, NdotH, 0.0);
perlight float   NdotH2  = NdotHs * NdotHs;
perlight float   NdotH4  = NdotH2 * NdotH2;
perlight float   NdotH8  = NdotH4 * NdotH4;
perlight float3 spec      = NdotH8 * shadow * s;

// Combine
perlight float3 C = diff + spec;
return integrate(rgb(Cl) * C) + a;
} // lightmodel_bumps
```

Compiler output

	RGB	ALPHA
0	T3.rgb = (2*[T2.rgb]-1) dot (2*[T3.rgb]-1) T2.rgb = (2*[T2.rgb]-1) dot (2*[V0.rgb]-1)	S0.a = T3.b
1	T0.rgb = T0.rgb + T1.rgb * (1-[T0.aaa])	V0.a = (S0.a < 0.5) ? [Z0.a] : [T3.b]
2	T0.rgb = T0.rgb * T1.aaa	V0.a = V0.a * V0.a T0.a = T2.b
3	T1.rgb = 0.5*T0.rgb	V0.a = V0.a * V0.a
4	V0.rgb = T1.rgb * [T0.aaa] T1.rgb = V0.aaa * V0.aaa	T1.a = 4*((2*[V0.b]-1) + (2*[V0.b]-1))
5	V0.rgb = V0.rgb * [T1.aaa]	V0.a = T1.b * T1.a
6	L0.rgb = {0.300000, 0.300000, 0.300000} V0.rgb = V0.rgb * T2.aaa + V0.aaa * L0.rgb	
7	L0.rgb = {0.400000, 0.400000, 0.400000} V0.rgb = V1.rgb * V0.rgb + T0.rgb * L0.rgb	
F	OUT.rgb = [V0.rgb]	OUT.a = (1-[Z0.a])

Code for first RGB combiner

```
T3.rgb = (2*[T2.rgb]-1) dot (2*[T3.rgb]-1)
T2.rgb = (2*[T2.rgb]-1) dot (2*[V0.rgb]-1)
```

System demonstration

Demo Credits

Fish

Real-time demo: Ren Ng
Animation/Models: Xiaoyuan Tu, Homan Igehy, Gordon Stoll

Real-time "Textbook Strike"

Real-time demo: Pradeep Sen
Original Scene: Tom Porter
Animation data: Anselmo Lastra, Lawrence Kesteloot, Fredrik Fatemi

Mouse Volume

Real-time demo: Ren Ng
Data Set: G. A. Johnson, G.P.Cofer, S.L. Gewalt, L.W. Hedlund, Duke Center for In Vivo Microscopy

Ear Volume

Real-time demo: Ren Ng
Data Set: Klaus Engel's web page

The compilation task

Generate HW code for a *basic block*

Basic block is represented by a DAG

Compilation is NP – use heuristic algorithms

Five stages in compiler

- Extract texture-shader operations
- Rewrite DAG to use HW operations
- Select instructions
- Allocate pipeline-input registers
- Schedule instructions and allocate registers

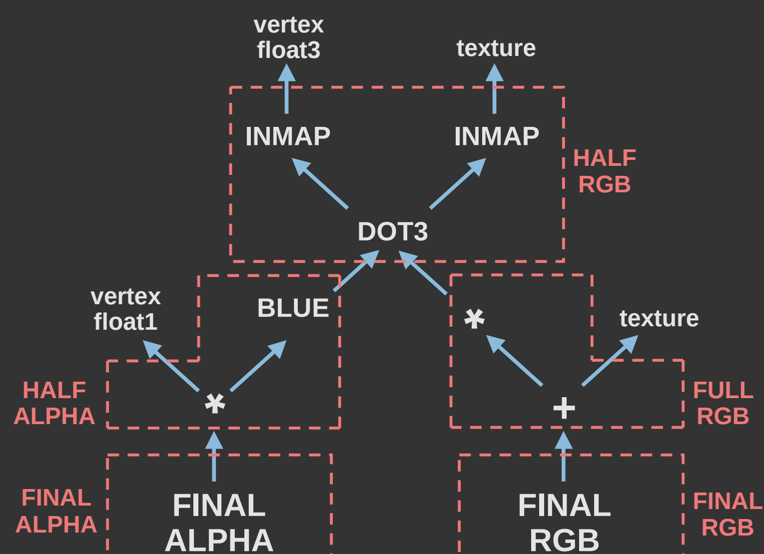
We focused on compiling to a single pass

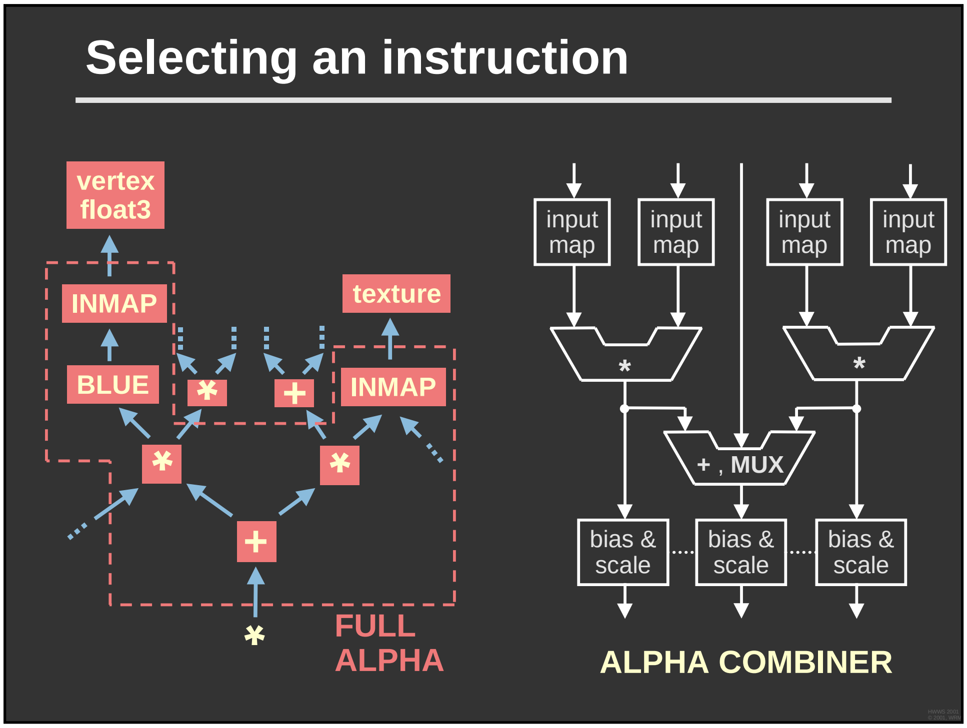
2. Rewrite DAG to use HW ops

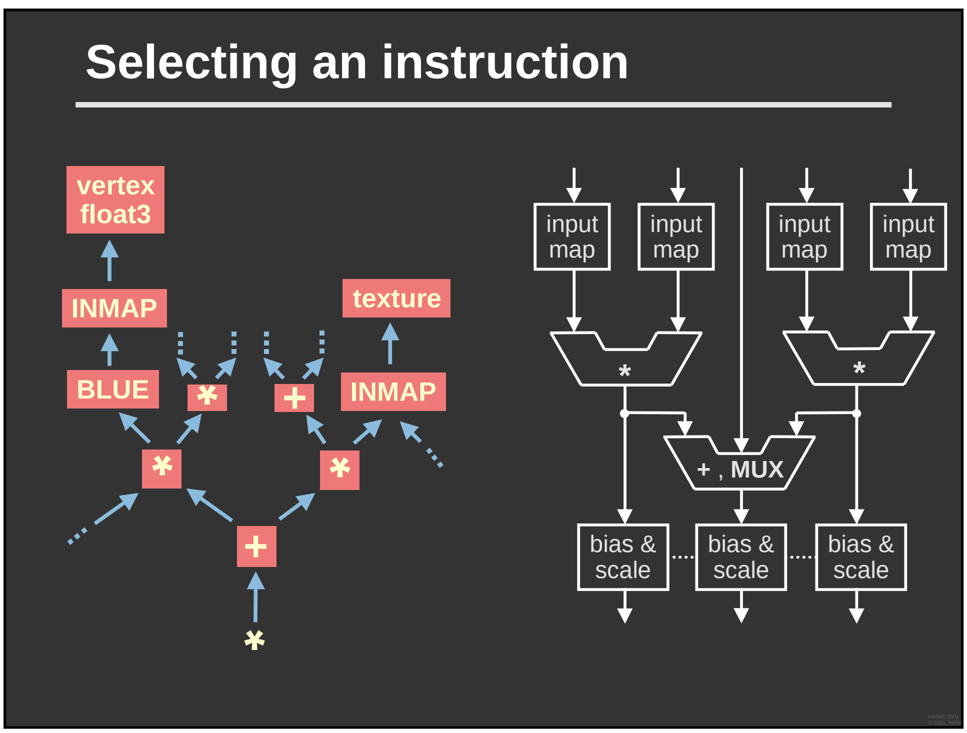
From…		To…
float4 * float4	→	float3 * float3 float1 * float1
X **DOT** Y	→	**BLUE (** X **DOT3** Y **)**
2 * (X - 0.5)	→	**INMAP(X, expand_normal)**

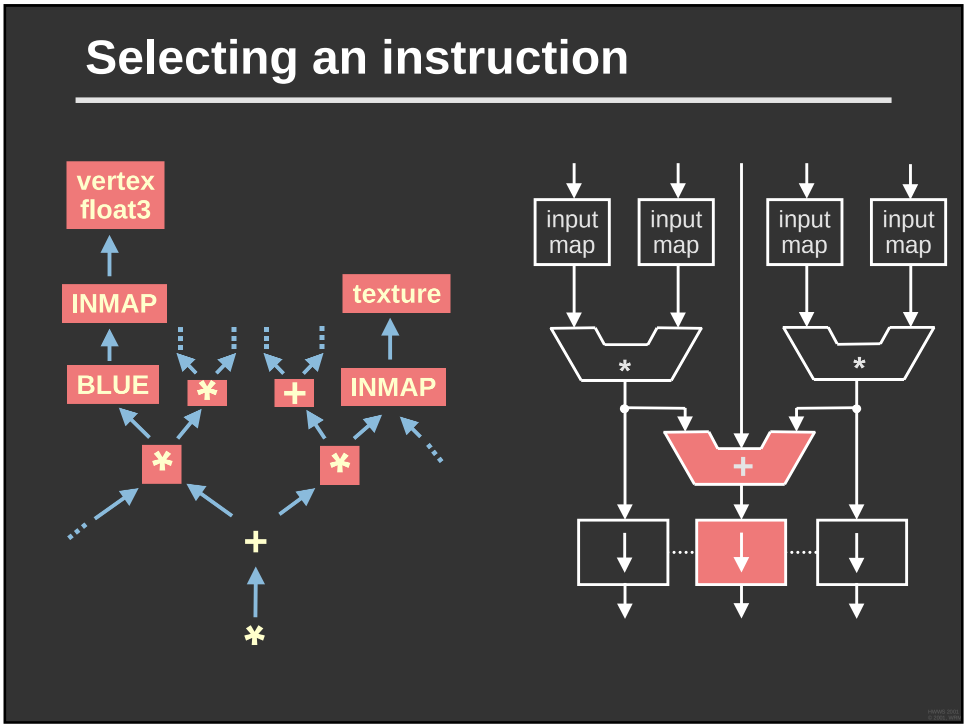
3. Select instructions

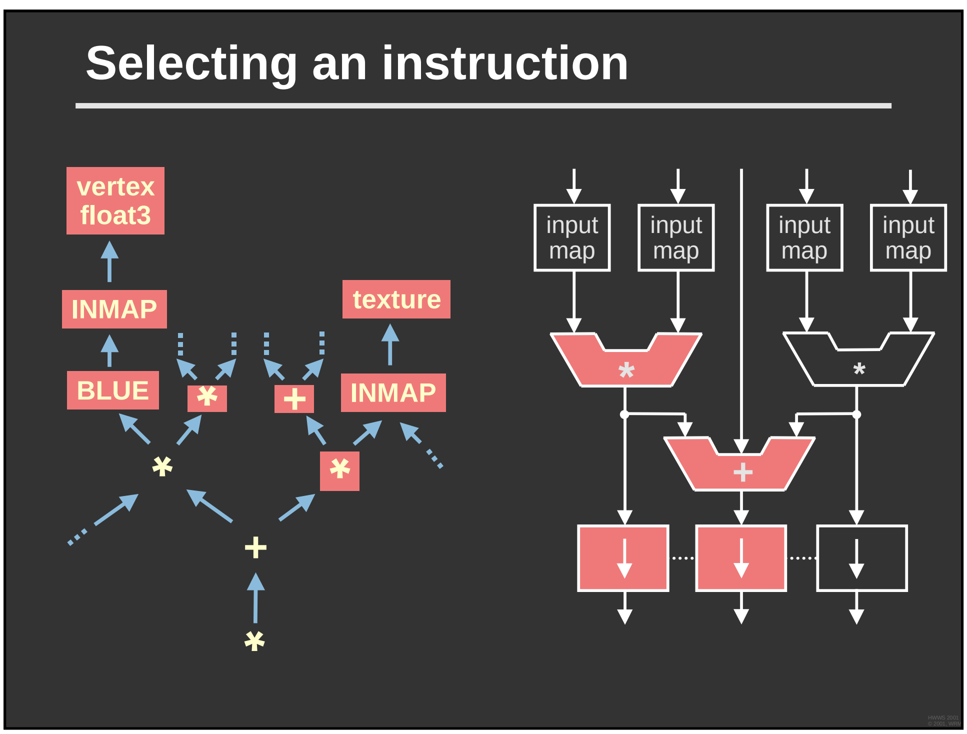
DAG traversal maps ops to full or half combiners

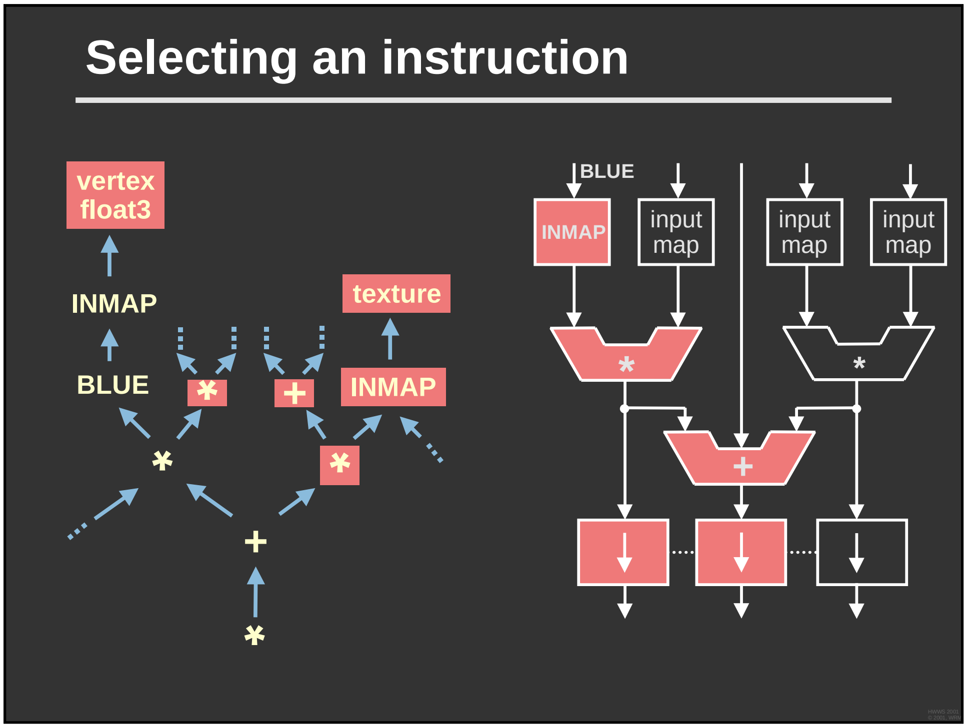


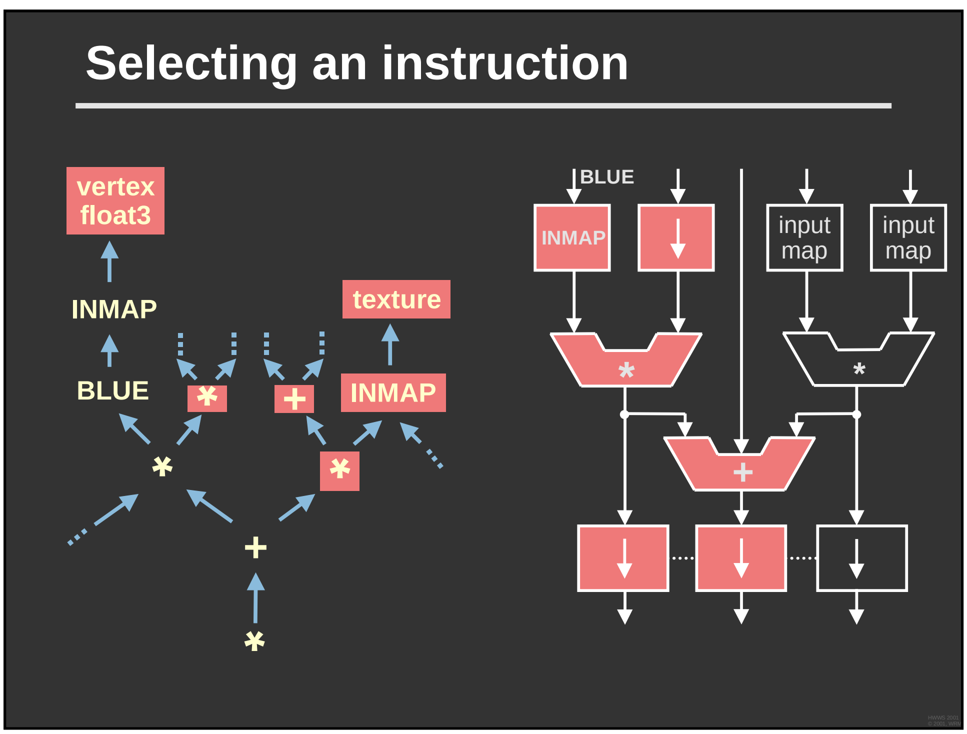

Selecting an instruction
vertex
float3
INMAP
BLUE
texture
INMAP
FULL
ALPHA
input
map
input
map
input
map
input
map
+ , MUX
bias &
scale
bias &
scale
bias &
scale
ALPHA COMBINER

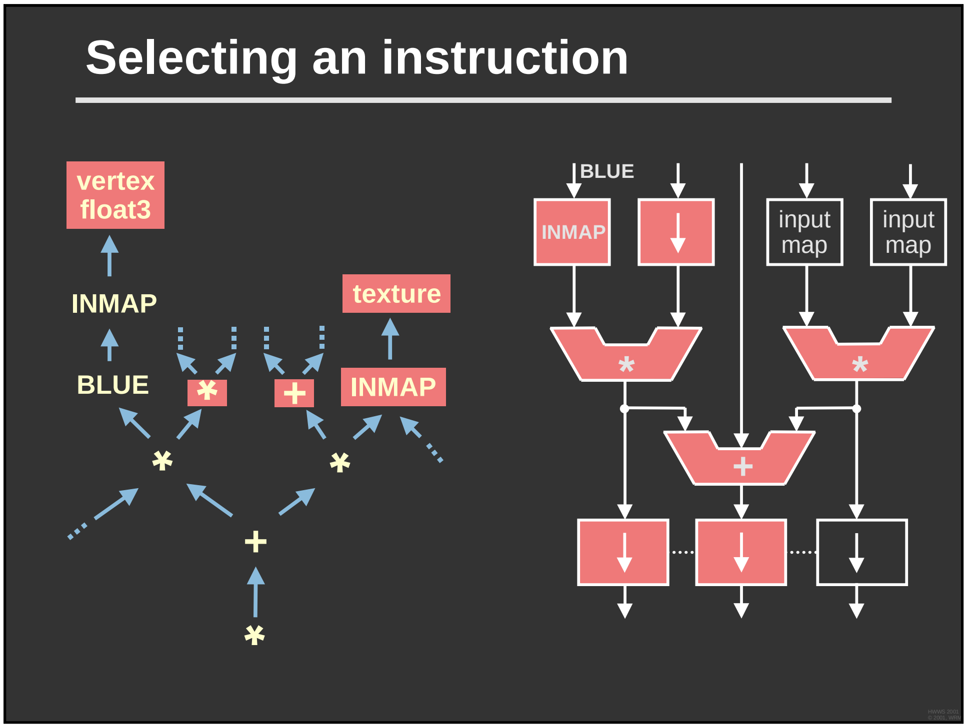

Selecting an instruction
vertex
float3
INMAP
BLUE
texture
INMAP
input
map
input
map
input
map
input
map
+ , MUX
bias &
scale
bias &
scale
bias &
scale

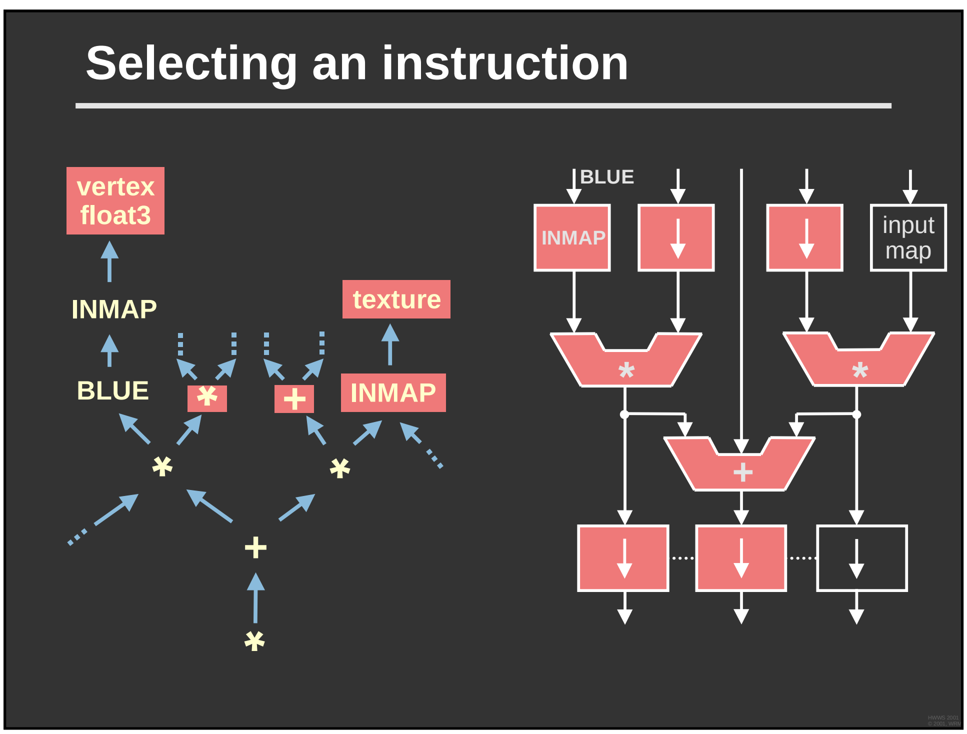

Selecting an instruction
vertex float3
INMAP
BLUE
texture
INMAP
input map
input map
input map
input map

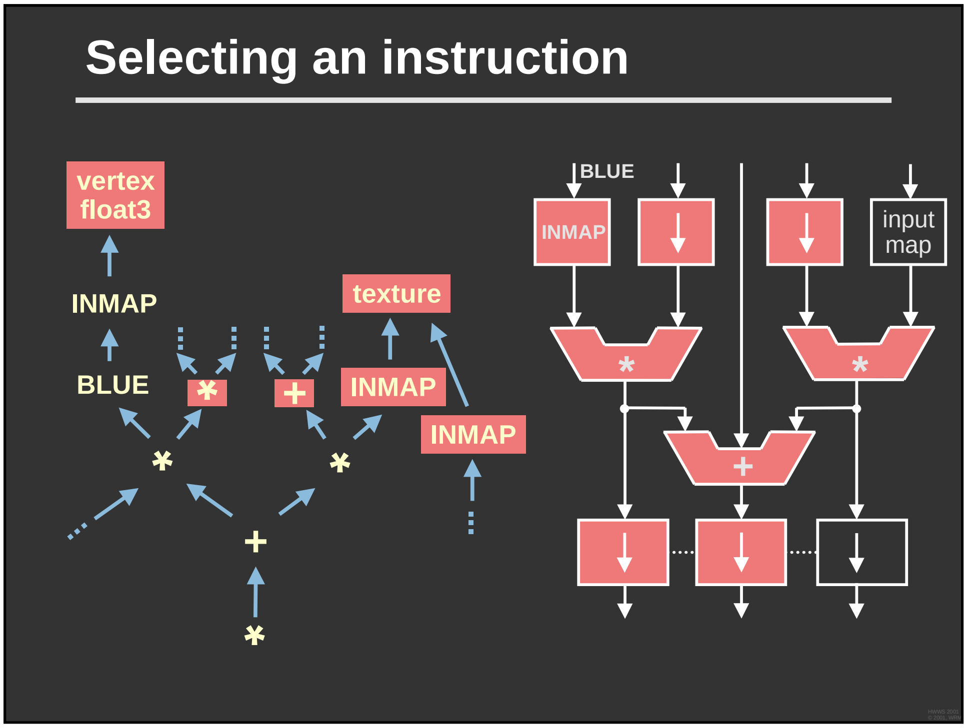

Selecting an instruction
vertex float3
INMAP
BLUE
texture
INMAP
input map
input map
input map
input map

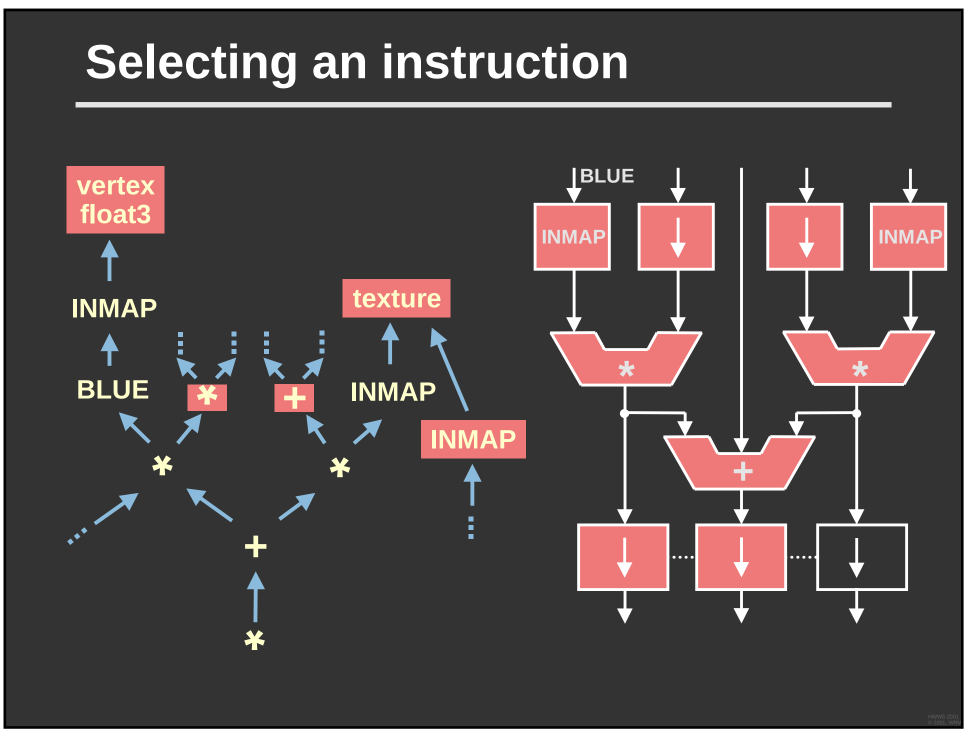

Selecting an instruction
vertex
float3
INMAP
BLUE
texture
INMAP
+
BLUE
INMAP
input
map
input
map
input
map
*
*
+


Selecting an instruction
vertex
float3
INMAP
BLUE
texture
INMAP
+
BLUE
INMAP
input
map
input
map
*
*
+

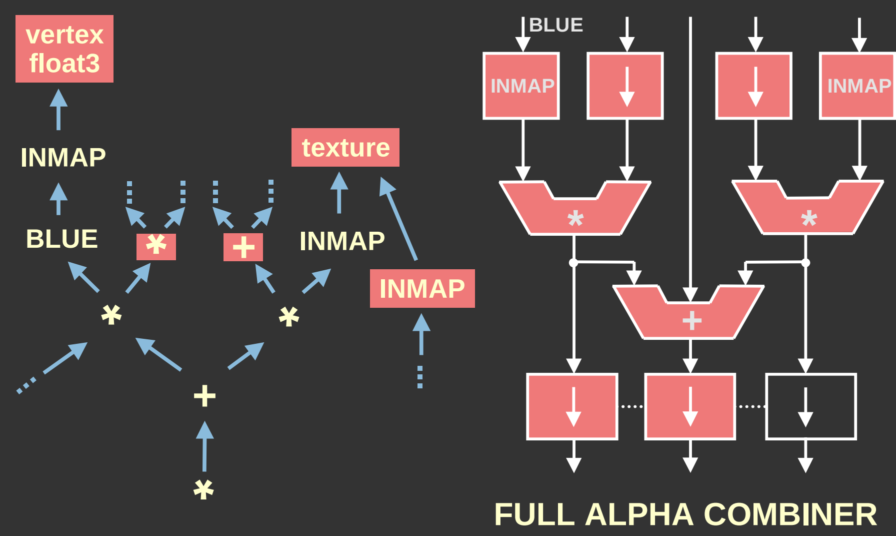

Selecting an instruction
vertex
float3
INMAP
BLUE
texture
INMAP
BLUE
INMAP
input
map
input
map


Selecting an instruction
vertex
float3
INMAP
BLUE
texture
INMAP
BLUE
INMAP
input
map


Selecting an instruction
vertex
float3
INMAP
BLUE
texture
INMAP
INMAP
BLUE
INMAP
input
map


Selecting an instruction
vertex
float3
INMAP
BLUE
texture
INMAP
INMAP
BLUE
INMAP
INMAP

Selecting an instruction



4. Allocate pipeline-input registers

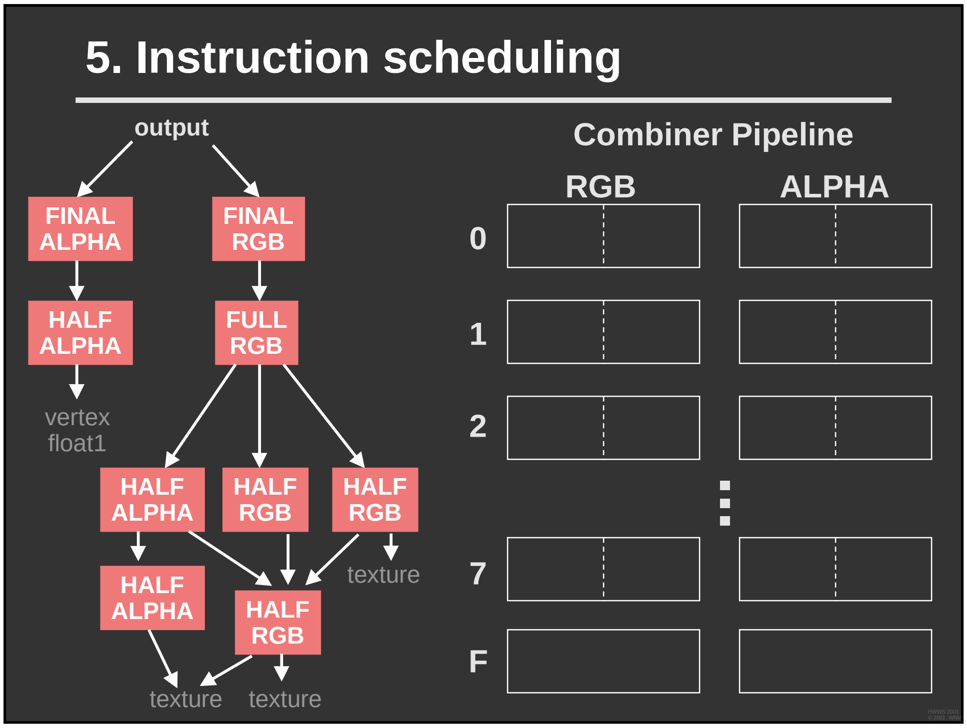
Pipeline inputs consist of:

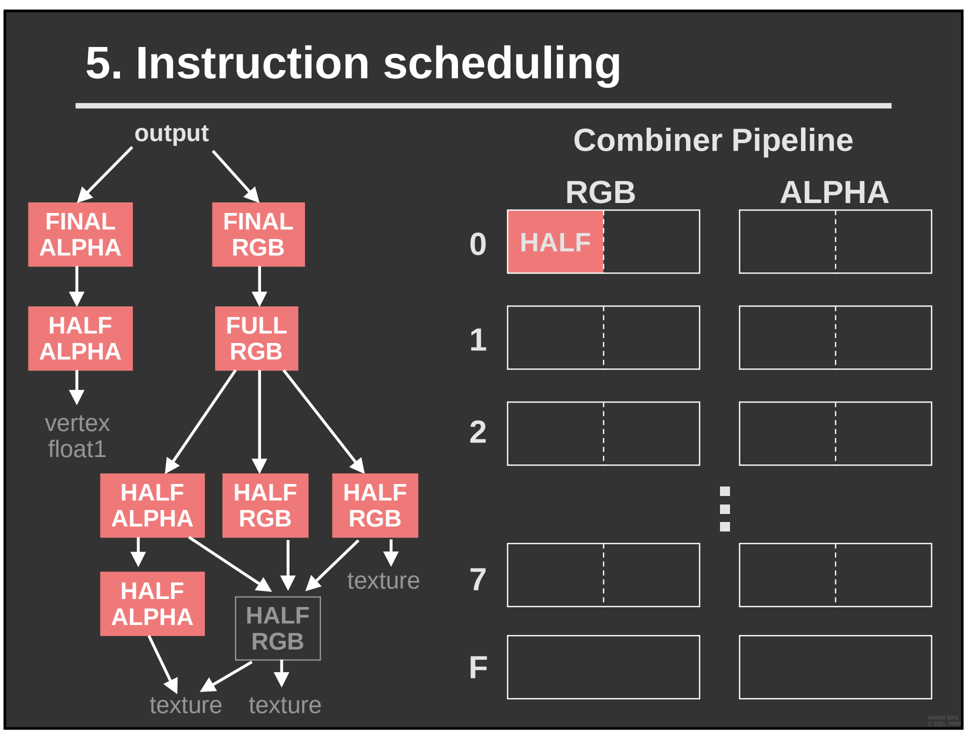
- textures
- interpolants from vertex values
- constants and “primitive group” values

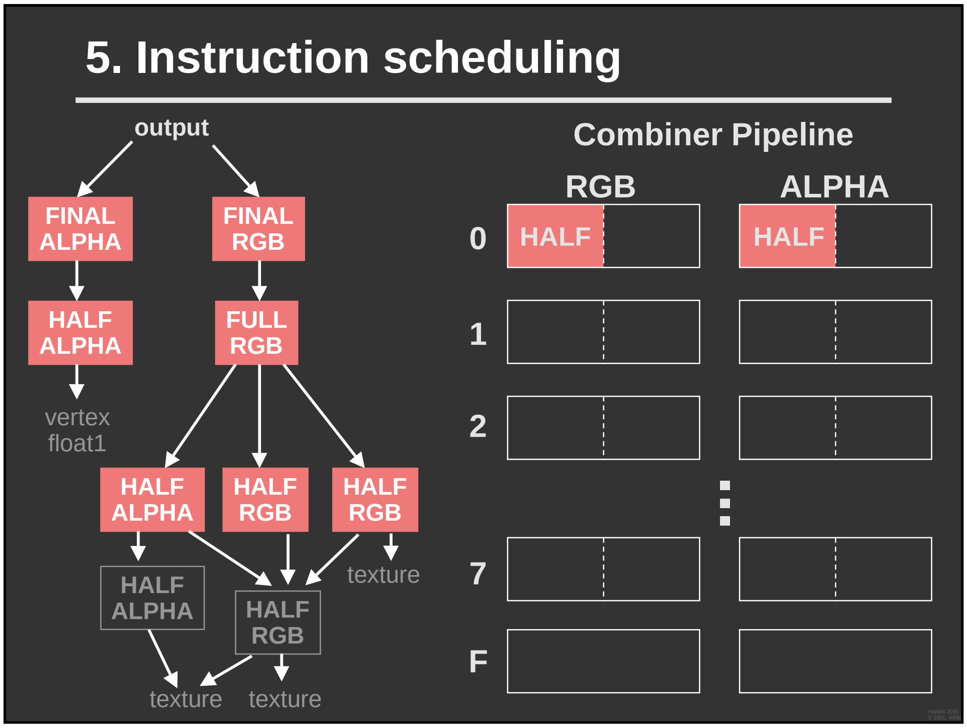
Use a greedy algorithm -- do “hardest” cases first

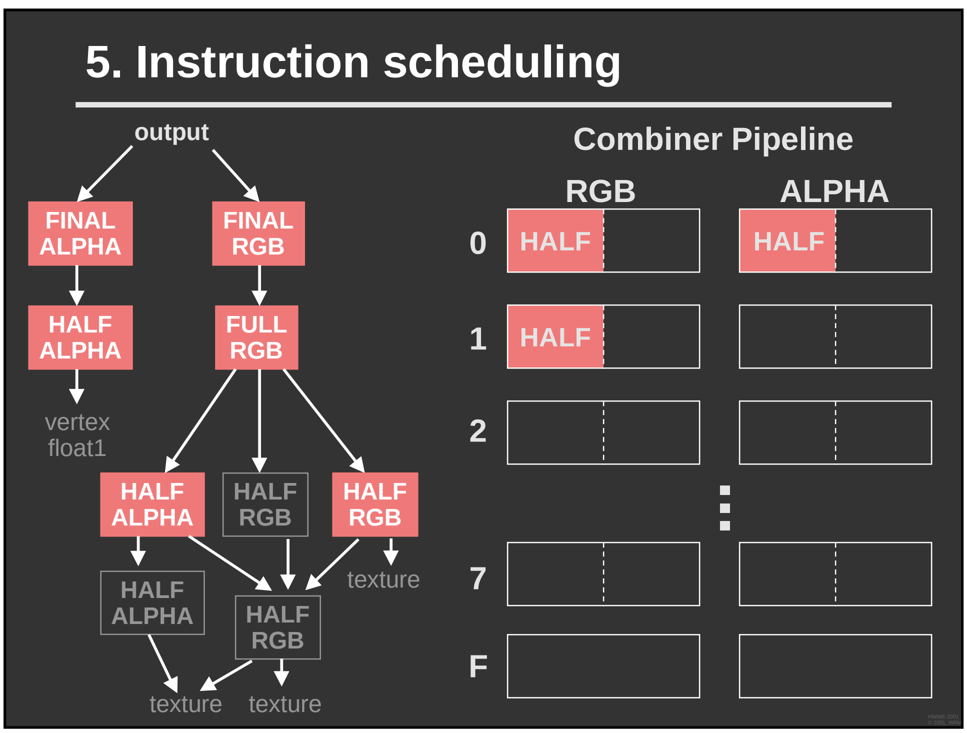
Some capabilites:

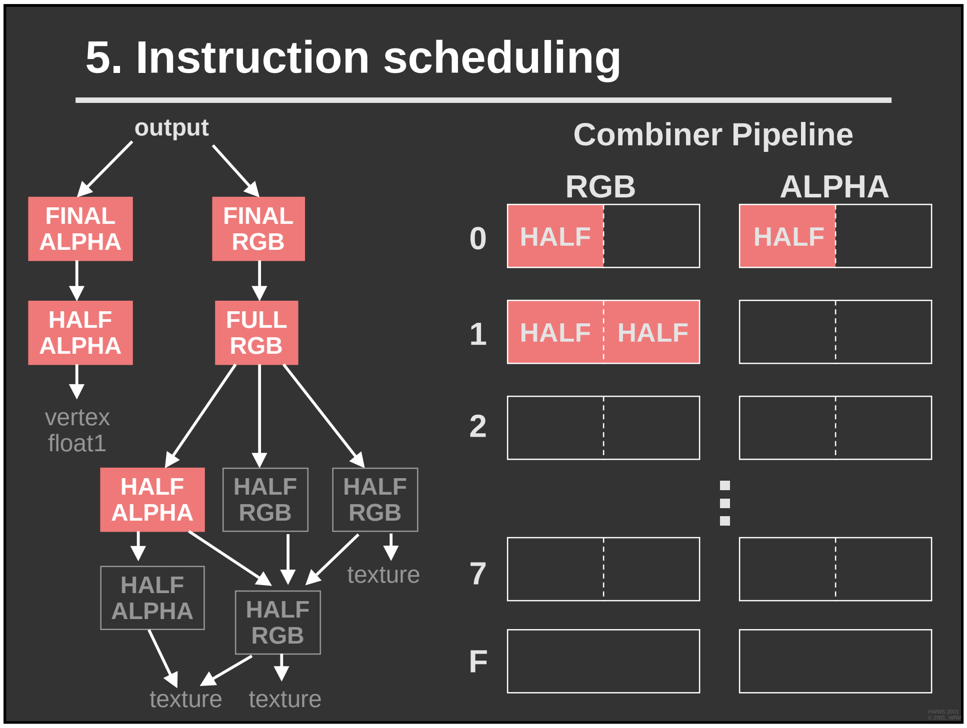
- pack unrelated 3-vector and scalar into RGBA
- put scalar in RGB
- use PASSTHRU texture for vertex interpolants

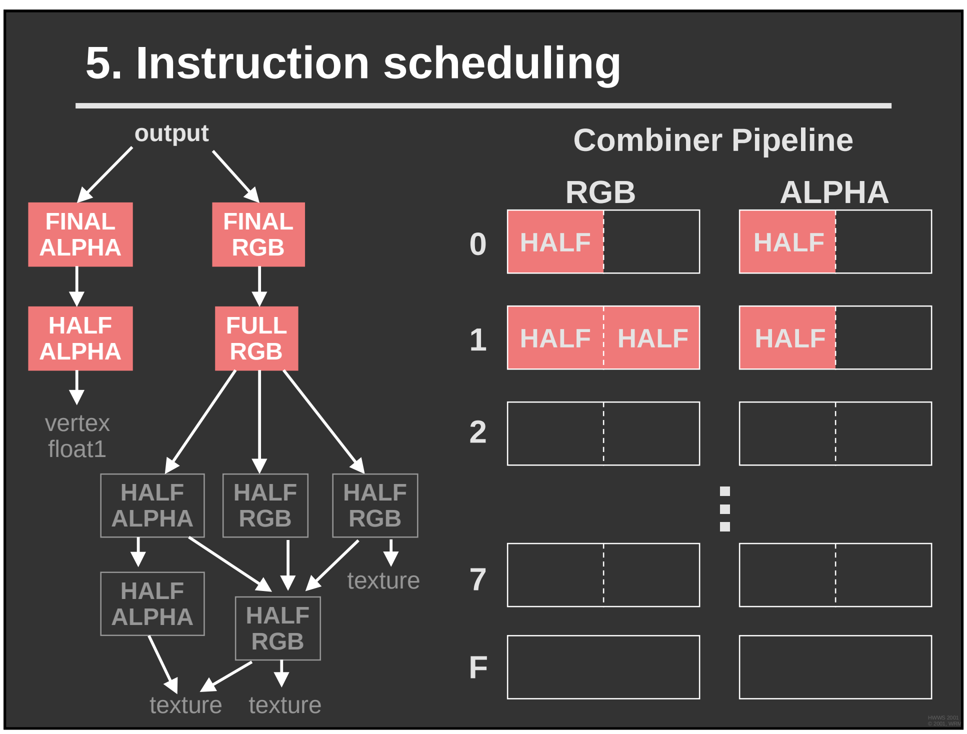

5. Instruction scheduling
output
FINAL ALPHA
FINAL RGB
HALF ALPHA
FULL RGB
vertex float1
HALF ALPHA
HALF RGB
HALF RGB
HALF ALPHA
HALF RGB
texture
texture
texture
Combiner Pipeline
RGB
ALPHA
0
1
2
7
F

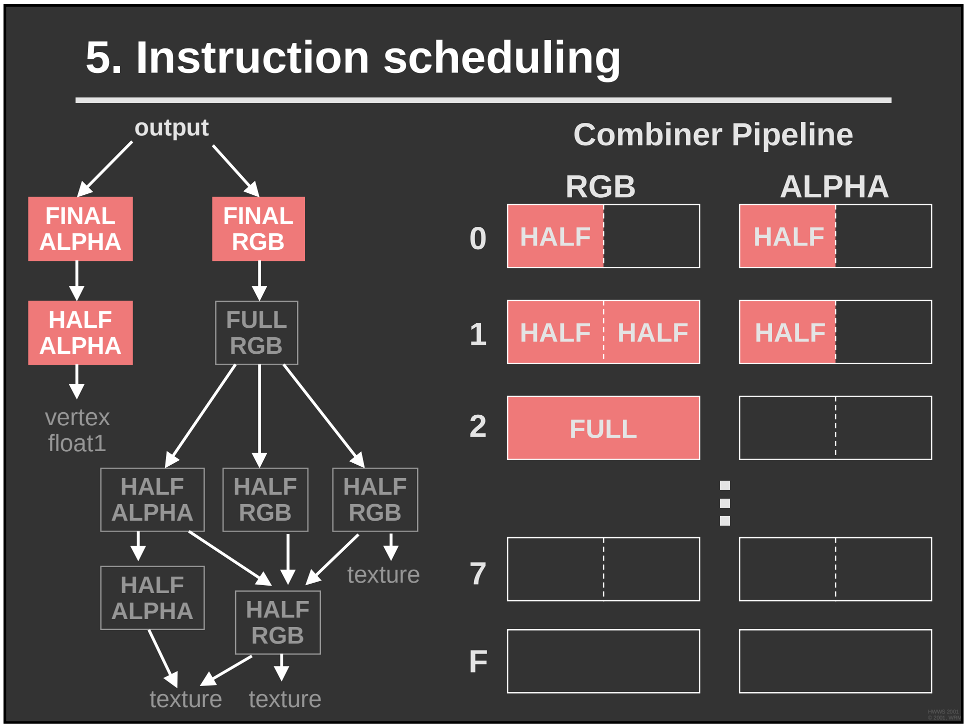

5. Instruction scheduling
output
FINAL ALPHA
FINAL RGB
HALF ALPHA
FULL RGB
vertex float1
HALF ALPHA
HALF RGB
HALF RGB
HALF ALPHA
HALF RGB
texture
texture
texture
Combiner Pipeline
RGB
ALPHA
0
HALF
1
2
7
F

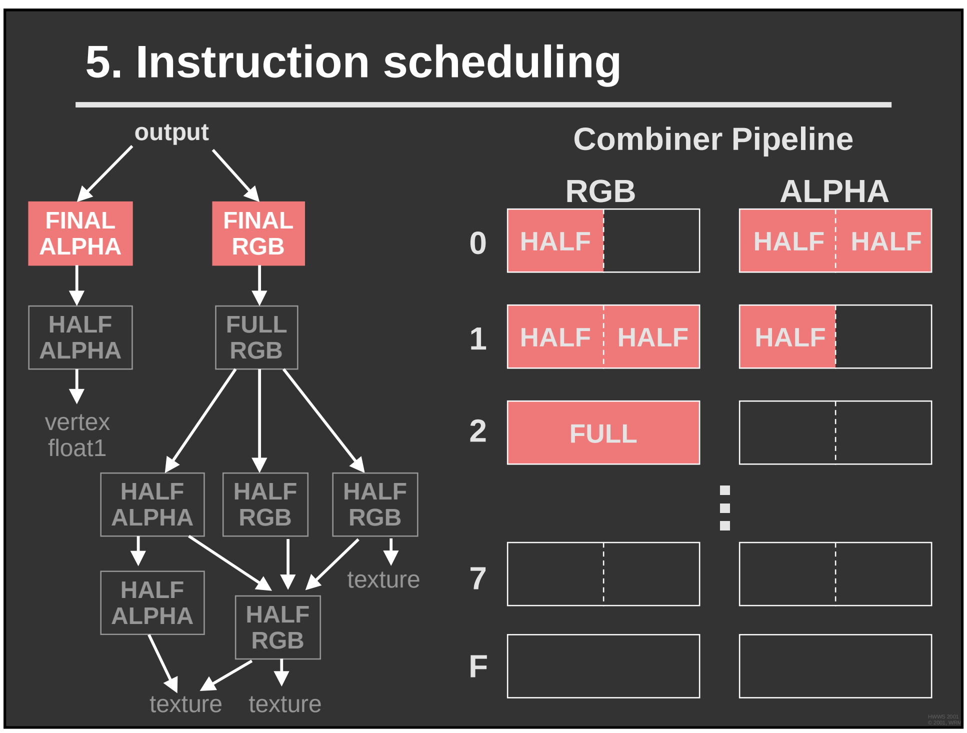

5. Instruction scheduling
output
FINAL ALPHA
FINAL RGB
HALF ALPHA
FULL RGB
vertex float1
HALF ALPHA
HALF RGB
HALF RGB
HALF ALPHA
HALF RGB
texture
texture
texture
Combiner Pipeline
RGB
ALPHA
0
HALF
HALF
1
2
7
F

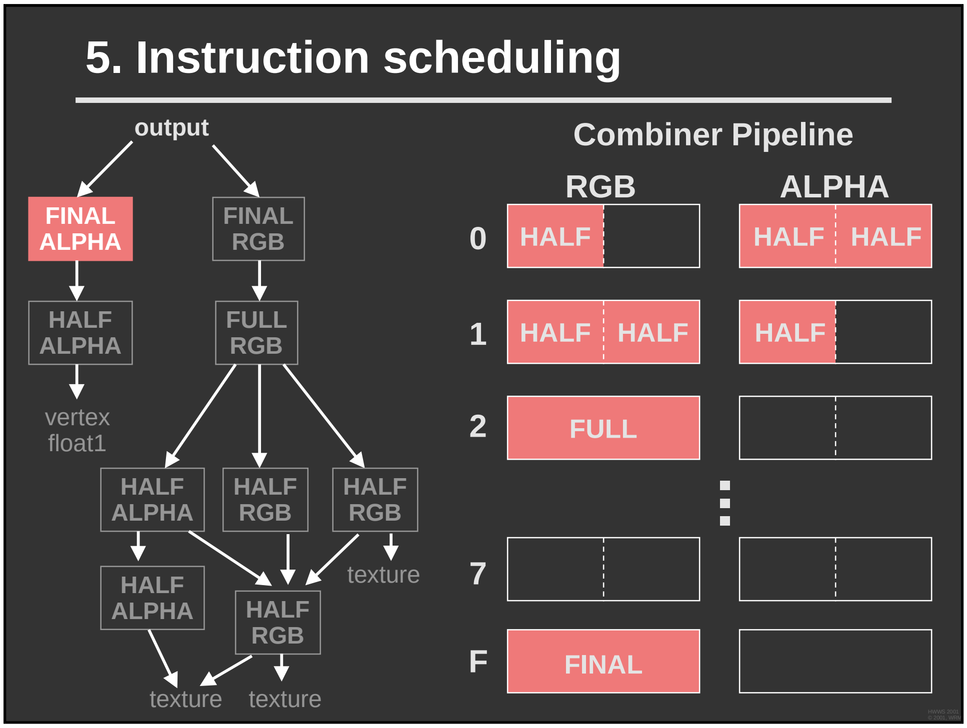

5. Instruction scheduling
output
FINAL ALPHA
FINAL RGB
HALF ALPHA
FULL RGB
vertex float1
HALF ALPHA
HALF RGB
HALF RGB
HALF ALPHA
HALF RGB
texture
texture
texture
Combiner Pipeline
RGB
ALPHA
0
HALF
HALF
1
HALF
2
7
F

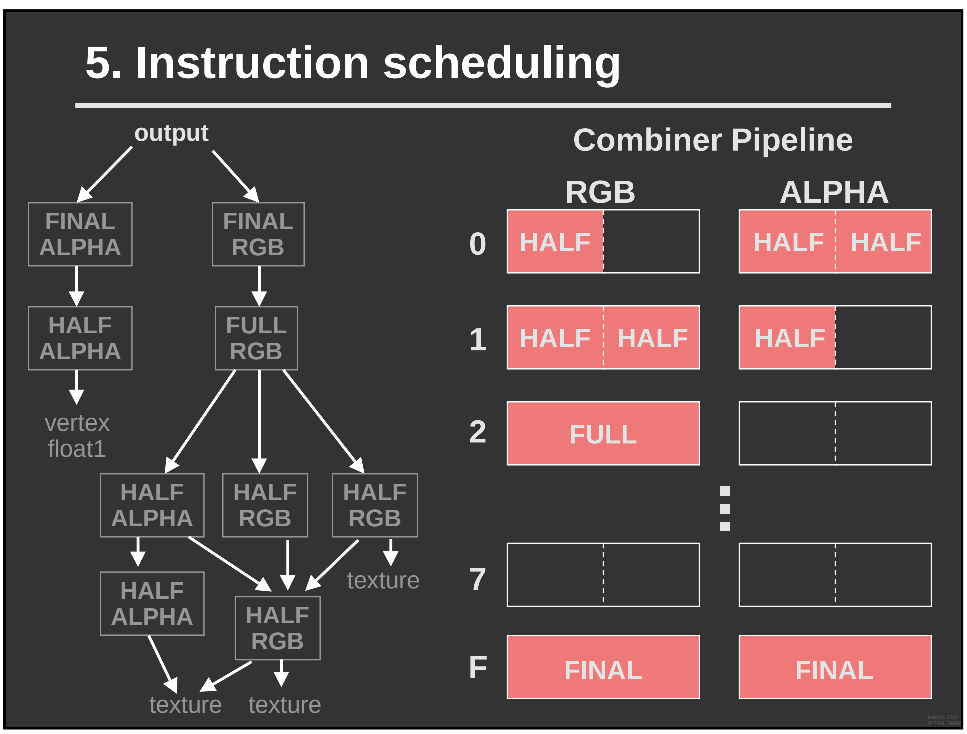

5. Instruction scheduling
output
FINAL ALPHA
FINAL RGB
HALF ALPHA
FULL RGB
vertex float1
HALF ALPHA
HALF RGB
HALF RGB
texture
HALF ALPHA
HALF RGB
texture
texture
Combiner Pipeline
RGB
ALPHA
0
HALF
HALF
1
HALF
HALF
2
7
F


5. Instruction scheduling
output
FINAL ALPHA
FINAL RGB
HALF ALPHA
FULL RGB
vertex float1
HALF ALPHA
HALF RGB
HALF RGB
texture
HALF ALPHA
HALF RGB
texture
texture
Combiner Pipeline
RGB
ALPHA
0
HALF
HALF
1
HALF
HALF
HALF
2
7
F


5. Instruction scheduling
output
FINAL ALPHA
FINAL RGB
HALF ALPHA
FULL RGB
vertex float1
HALF ALPHA
HALF RGB
HALF RGB
texture
HALF ALPHA
HALF RGB
texture
texture
Combiner Pipeline
RGB
ALPHA
0
HALF
HALF
1
HALF
HALF
HALF
2
FULL
7
F


5. Instruction scheduling
output
FINAL ALPHA
FINAL RGB
HALF ALPHA
FULL RGB
vertex float1
HALF ALPHA
HALF RGB
HALF RGB
texture
HALF ALPHA
HALF RGB
texture
texture
Combiner Pipeline
RGB
ALPHA
0
HALF
HALF
HALF
1
HALF
HALF
HALF
2
FULL
7
F


5. Instruction scheduling
output
FINAL ALPHA
FINAL RGB
HALF ALPHA
FULL RGB
vertex float1
HALF ALPHA
HALF RGB
HALF RGB
HALF ALPHA
HALF RGB
texture
texture
texture
Combiner Pipeline
RGB
ALPHA
0
HALF
HALF
HALF
1
HALF
HALF
HALF
2
FULL
7
F
FINAL


5. Instruction scheduling
output
FINAL ALPHA
FINAL RGB
HALF ALPHA
FULL RGB
vertex float1
HALF ALPHA
HALF RGB
HALF RGB
HALF ALPHA
HALF RGB
texture
texture
texture
Combiner Pipeline
RGB
ALPHA
0
HALF
HALF
HALF
1
HALF
HALF
HALF
2
FULL
7
F
FINAL
FINAL

Compiler generates efficient code

Example: Bowling pin shader

- Initially 8 combiners
- Can reduce to 7 by using compiled code to guide source-code changes
- Can't do any better by hand – this is typical

What the compiler can't do:

- Reorder mathematical operations
- Reorganize textures (e.g. join RGB with A)
- Design algorithms that map well to combiners

Key lessons

Scalar computations are common!

- E.g. bump mapping, volume rendering
- Compiler often puts scalar ops in RGB
- Implications for future HW

Data types

- Make types orthogonal to operations
- Avoid fixed-point type proliferation – regcomb has [-1,1] [-2,2] [0,1] [0,4]
- Compiler can only partially hide messiness

Good resource balance in NV20

- We've hit limits on textures, interpolants, and instructions
- Always enough temporary registers

HW trends and compilation

Cleaner HW designs

- Fewer idiosyncrasies
- Cleaner data types

Evolution away from VLIW? (see DX8)

- Can get parallelism from multiple fragments
- Don't need instruction-level parallelism
- But… scalars in vector units still look VLIW

Continue to need two types of "register allocation"

Better HW support for resource virtualization

Summary

Shading compilers can produce efficient code

- Good performance without tuning
- Can perform final tuning in high-level language

Need tighter coupling between HW and compilers

- CPU designers learned this a long time ago
- It will happen in graphics too

Important Results

- Scalar computations are common
- Clean data types are critical

Acknowledgements

Users, Demo Writing, and Debugging

- Ren Ng, Pradeep Sen

Stanford Shading Group & Collaborators

- Pat Hanrahan, Svetoslav Tzvetkov, Pradeep Sen, Ren Ng, Eric Chan, Philipp Slusallek, Reinhard Wilhelm, John Owens, Ian Buck, David Ebert, Marc Levoy

Sponsors

- ATI, NVIDIA, SONY, Sun
- DARPA, DOE

Special thanks to NVIDIA

- Matt Papakipos, Mark Kilgard, Nick Triantos

More information on the web

http://graphics.stanford.edu/projects/shading

- Download system
(binary only, but includes linkable library)
- Copies of papers
- Language and API specs

Questions?