RenderMan compiler for the RPU Ray Tracing Hardware Architecture

Tomasz Węgrzanowski

Overview

- RPU hardware provides more power
  - power – making more things possible (or practical)
- More power – more complex shaders
- Hand coding no longer practical
- RenderMan Shading Language (RSL) is a de-facto standard language for writing shaders
- RSL compiler targeting RPU makes it possible to fully use power of the hardware

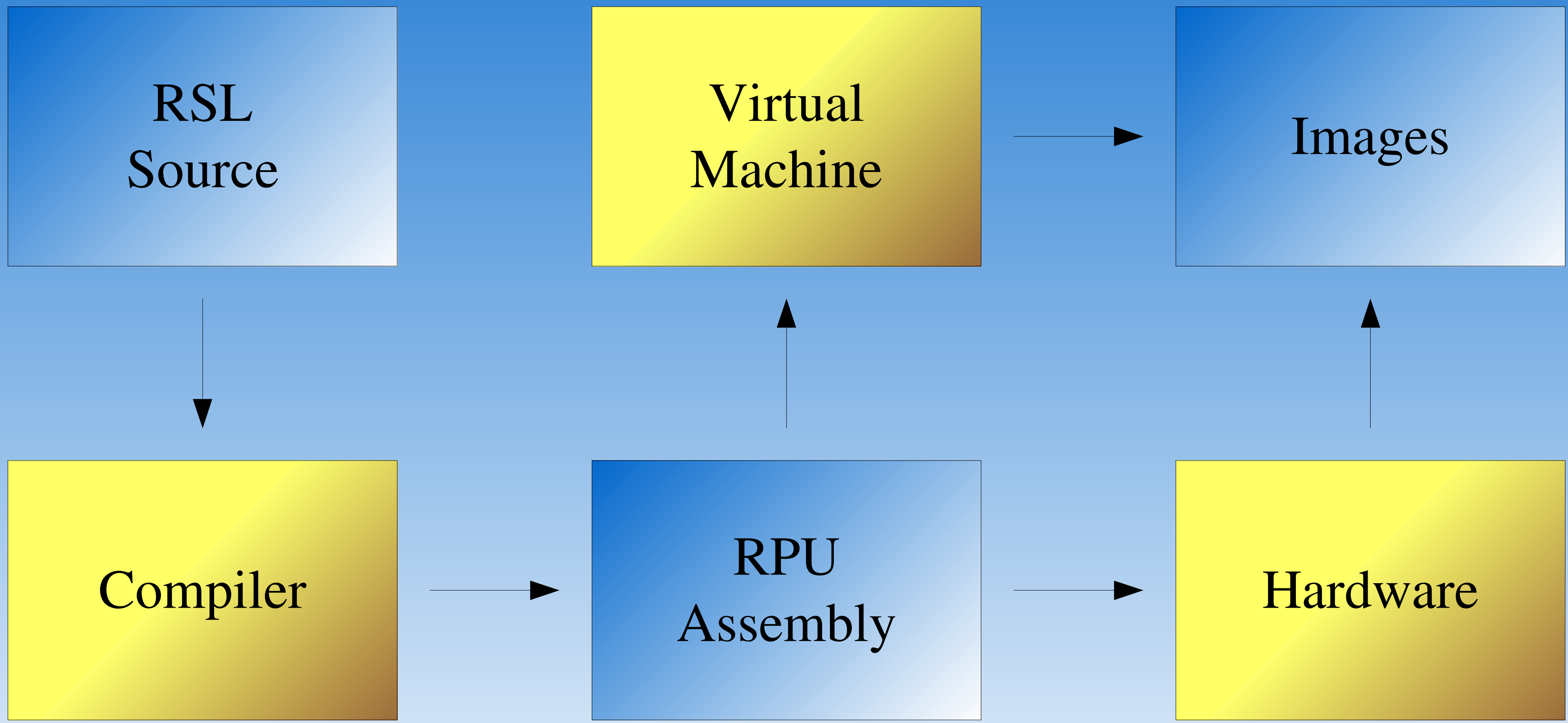
Compiler

- Full RSL support is infeasible
  - Regular expressions, memory allocation etc.
  - The theoretically feasible part is still too big
  - Power is important, not RSL compatibility
- Generated code must be highly efficient

Design

- Hardware is not easily available for testing
  - Even if it was, it would be difficult to debug
  - So a virtual machine is used
- Optimizing compiler is highly complex, with many corner cases
  - Bugs are very likely
  - Hundreds of tests to make bugs less likely
- Conservative compiler design

Information flow



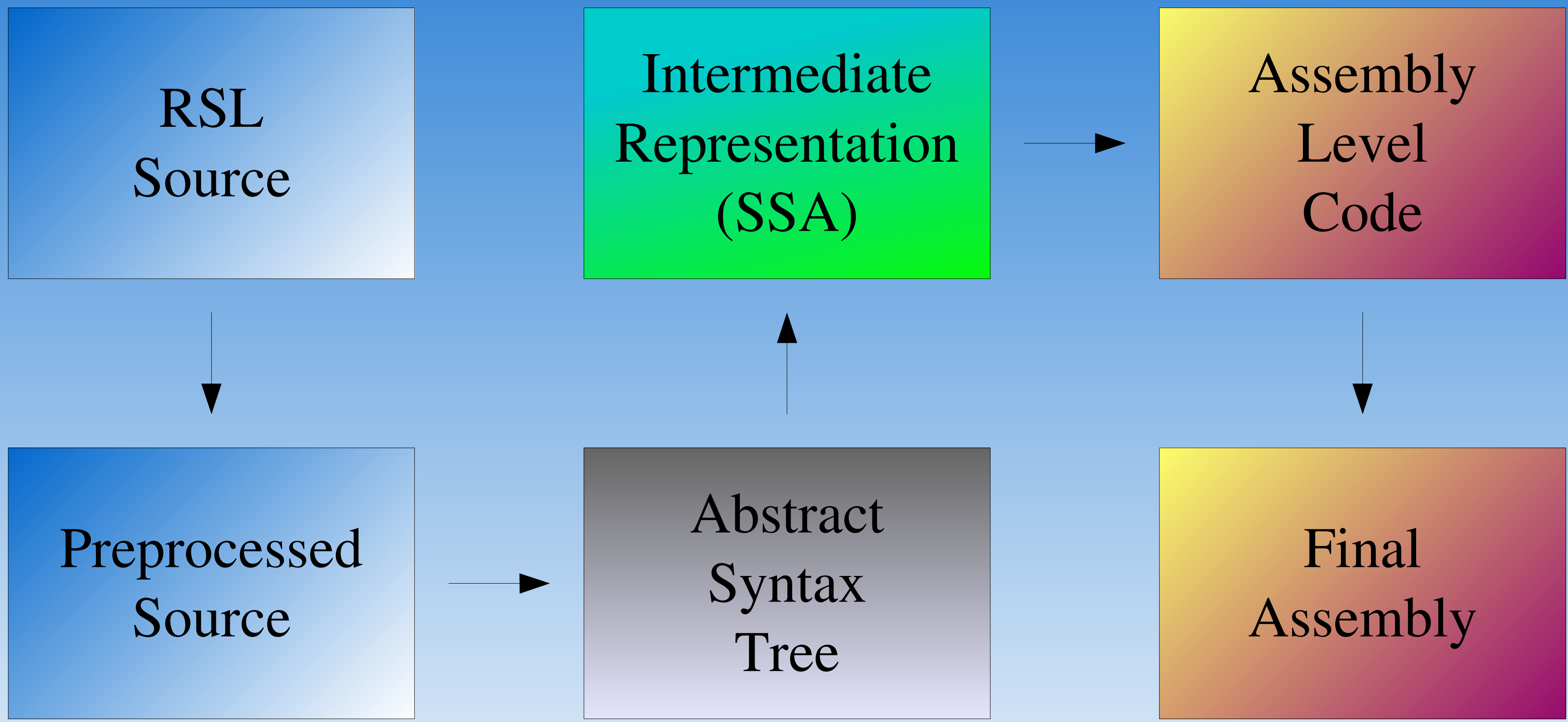
Virtual machine

- Virtual machine is a small Objective Caml library
- Multiple drivers use this library
  - Driver for debugging functions
  - Driver for debugging surface shaders
  - Driver that emulates tha complete hardware
  - Drivers for testing that VM works correctly
- Objective Caml is fast, expressive and doesn't segfault
  - Good for virtual machines

Assembler

- Very simple program that compiles RPU assembly instructions to virtual machine bytecode
- VM bytecode is not compatible with hardware machine code (different floating point format etc.)
- Separate assembler for compiling to actual hardware
- Implemented in Objective Caml
  - ocamllex+ocamlyacc for parsing
  - Objective Caml marshalling format for output to VM

Compiler


RSL Source
Intermediate Representation (SSA)
Assembly Level Code
Preprocessed Source
Abstract Syntax Tree
Final Assembly

Compiler big picture

- Compiler initially written in Objective Caml
  - Too much complexity
  - Refactoring helped little
- Now two programs
  - Parser and intermediate code generator in Objective Caml
  - Optimizer and assembly generator in Ruby
- Ruby handles complexity much better
  - Performance not an issue - shaders small, not real-time

Preprocessing

```
float f(float x) {
#ifdef FAST
    return x * (1-x*x/6);
#else
    return sin(x);
#endif
}
```

cpp →

```
float f(float x) {
    return x * (1-x*x/6);
}
```

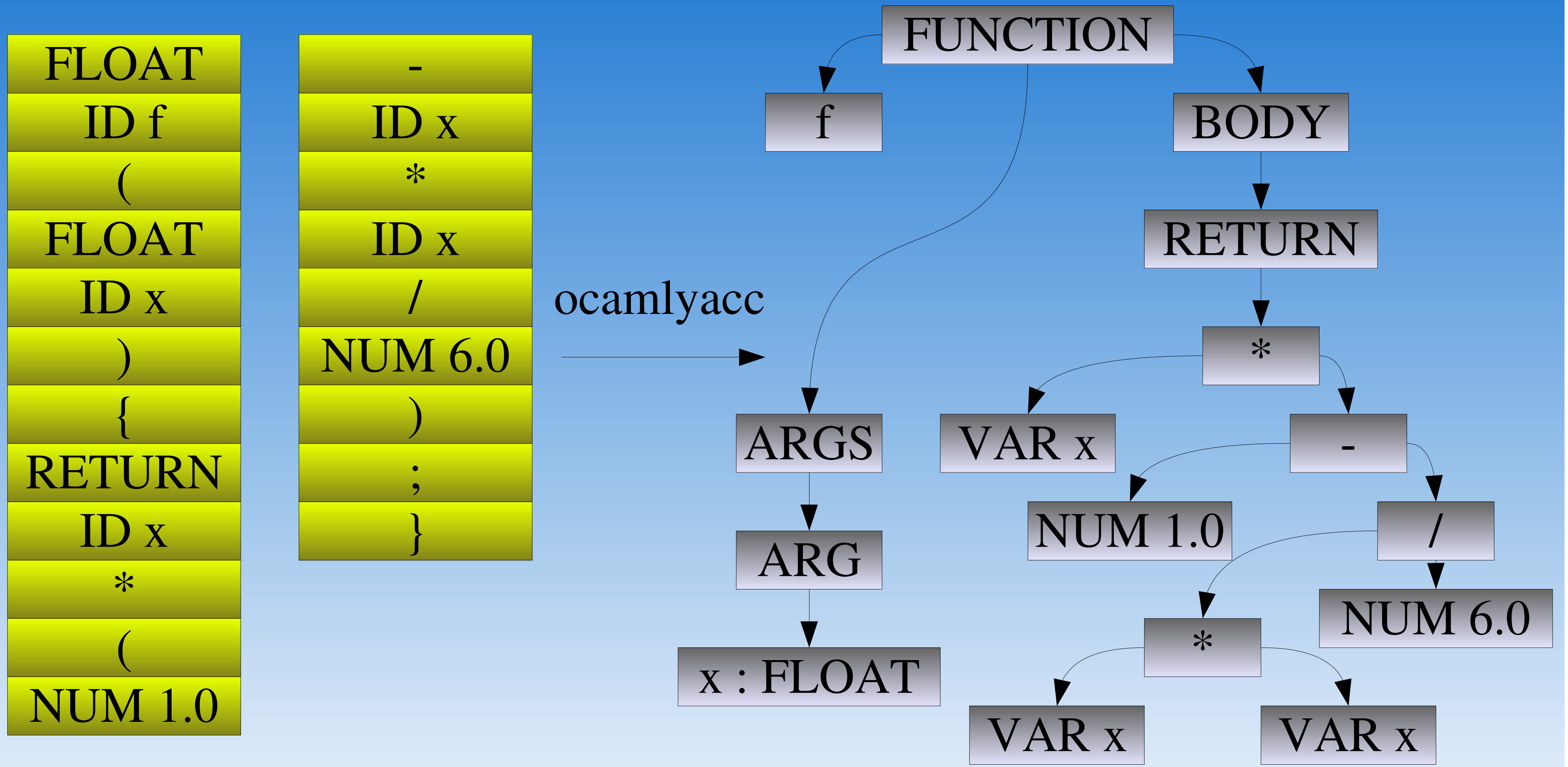
Lexing

```
float f(float x) {
    return x * (1-x*x/6);
}
```

ocamllex →

FLOAT	-
ID f	ID x
(	*
FLOAT	ID x
ID x	/
)	NUM 6.0
{	)
RETURN	;
ID x	}
*	
(	
NUM 1.0	

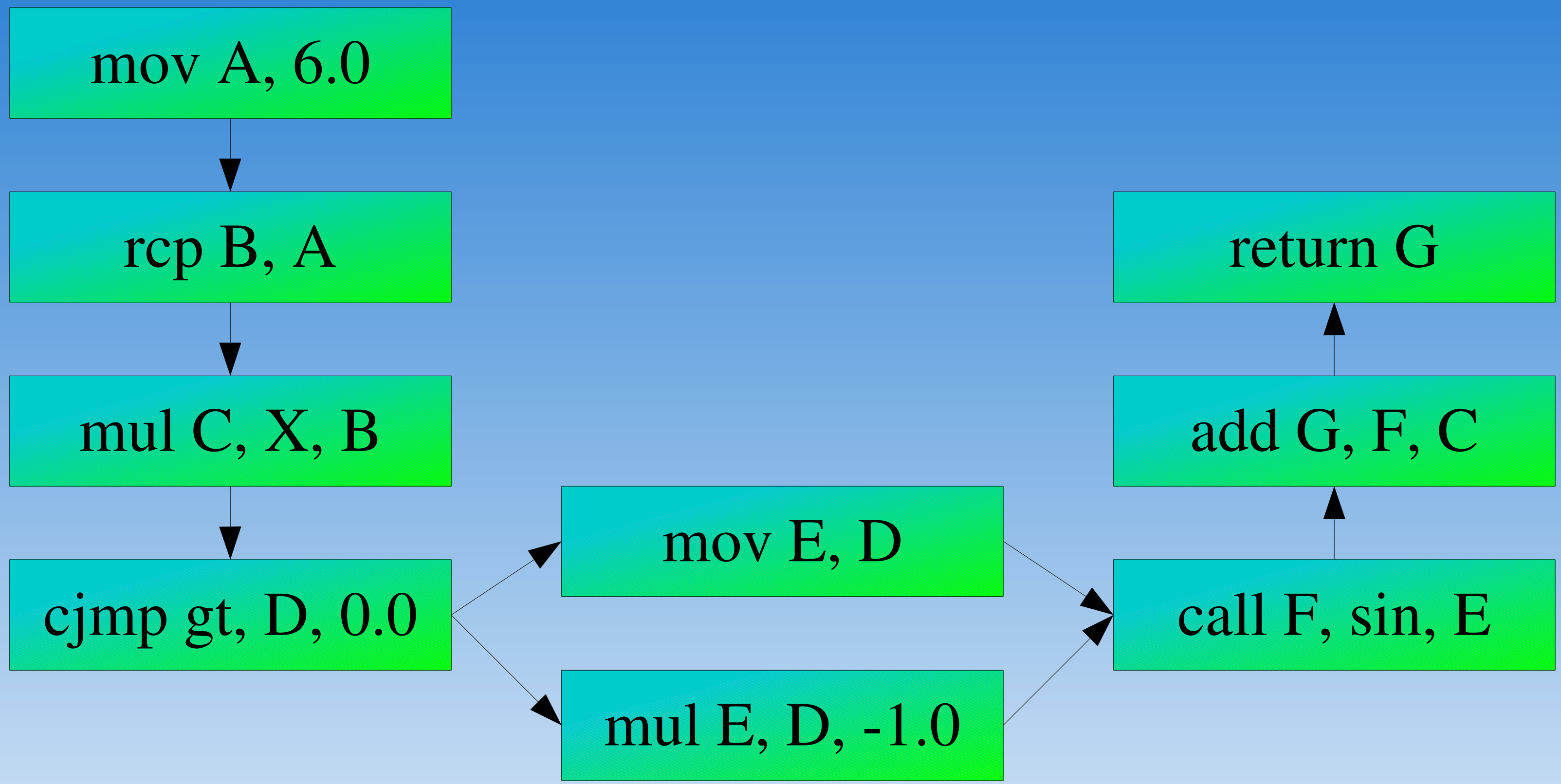
Parsing


FLOAT
ID f
(
FLOAT
ID x
)
{
RETURN
ID x
*
(
NUM 1.0
-
ID x
*
ID x
/
NUM 6.0
)
;
}
ocamlyacc
FUNCTION
f
BODY
RETURN
*
ARGS
ARG
x : FLOAT
VAR x
-
NUM 1.0
/
NUM 6.0
*
VAR x
VAR x

Parsing

- Parsing with ocamllex+ocamlyacc
  - Simple
  - Works
  - Horrible error reporting
- Not good enough for "production" compiler
  - No good parser generators for Objective Caml
  - ANTLR works with Ruby (and Java, Python, C++ etc.)
  - Hand-coded recursive-descent parser would be OK too

Intermediate Representation



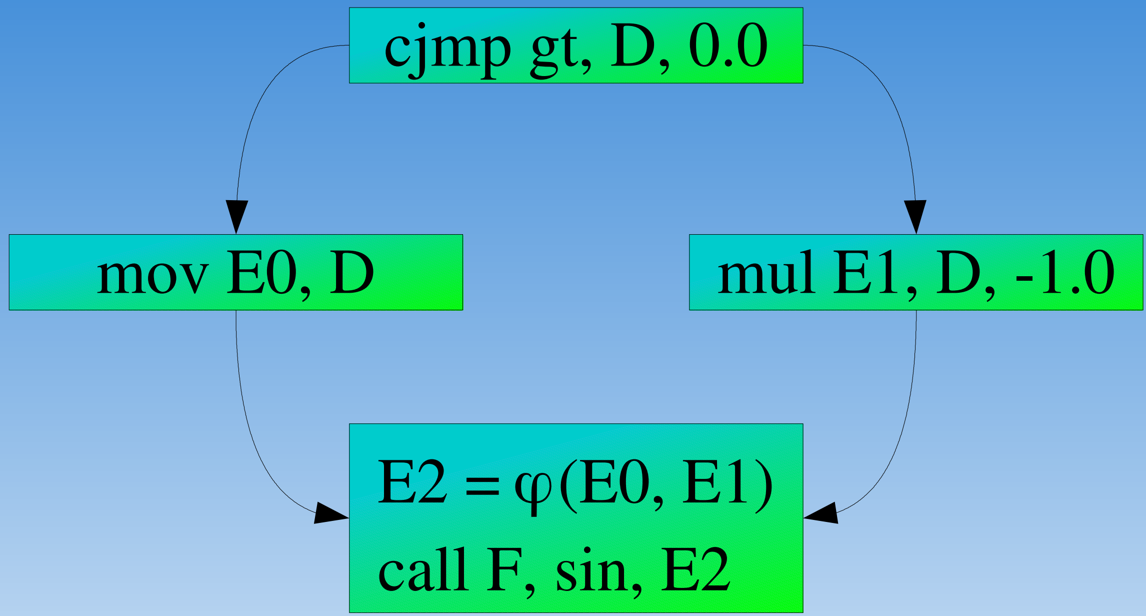
SSA

- Every variable has exactly one definition
- Variables with multiple definitions are split
- φ-functions inserted if multiple definitions reach
- Simplifies many optimizations
- Most modern compilers use SSA internally

SSA


mov A, 0.0
add C, B, A
mov A, 6.0
add D, B, A
Convert to SSA
mov A0, 0.0
add C, B, A0
mov A1, 6.0
add D, B, A1

SSA

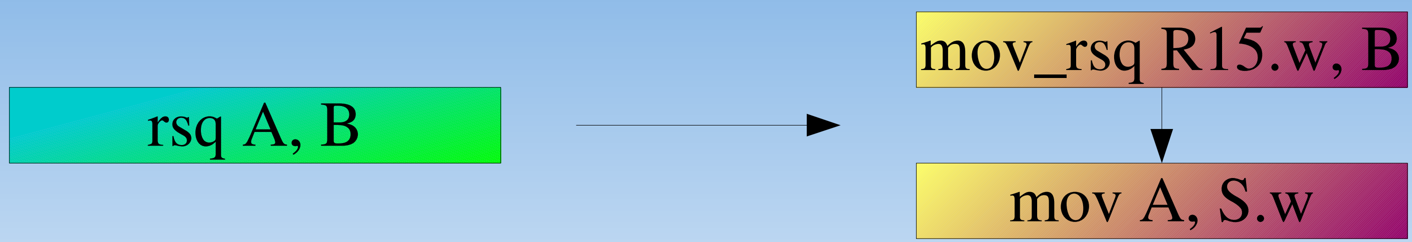


Benefits of SSA

- Many optimizations become extremely simple
- Copy propagation in SSA:
  - For every “mov A, B” anywhere
    - replace all uses of A by B
    - drop the instruction
- Without SSA copy propagation is rather complex
- Advanced optimizations become possible
  - Sparse Conditional Constant Propagation

Code generation

- Naively convert code from SSA to “almost-assembly”
  - Virtual registers
  - Stack handling, jumps later
  - not SSA any more (only local optimizations possible)



Forwarding

```
rsq B, A
mul D, C, B
```

```
mov_rsq R15.w, A
mov B, S.w
mul D, C, B
```

```
mov_rsq R15.w, A
mul D, C, S.w
```

More complex and less powerful than SSA copy propagation
Works well with RPU quirks like S.w (too low level for SSA)

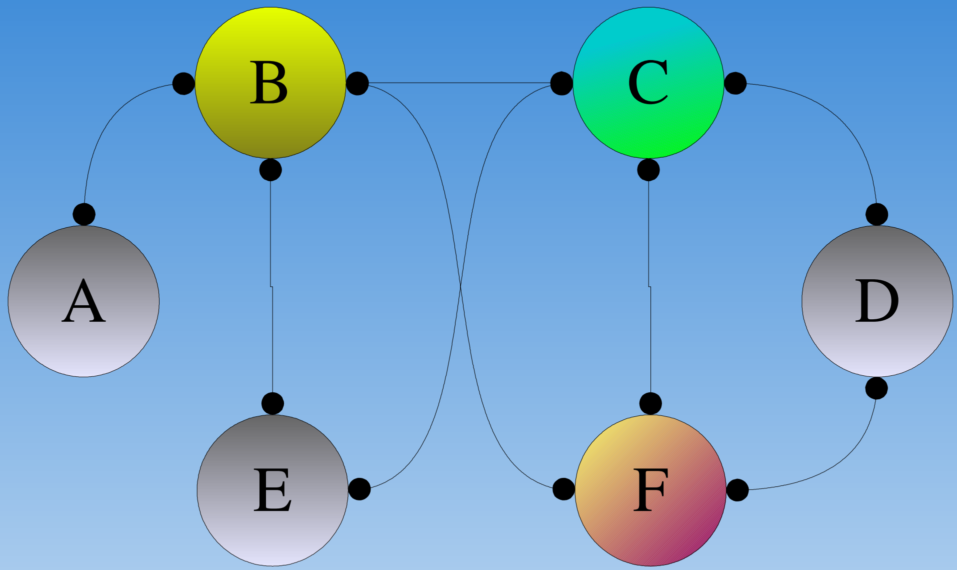
Opcode simplification

add A, B, 0.0	→	mov A, B
rcp A, 2.0	→	mov A, 0.5
add A, B, B	→	mov A, 2•B
mul A, B, -1.0	→	mov A, -B

Register allocation

- Easy because we have many registers
- Algorithm based on graph coloring
  - Each node is a variable
  - Edge between variables if both live at the same time
  - Each physical register has a different color
  - NP-Complete
    - efficient approximate algorithms exist
- We use heuristics for more efficient allocation

Register allocation



Remove registers with fewest conflict first

- A (1 conflict)
- D (2 conflicts)
- E (2 conflicts)
- B (2 conflicts)
- C (1 conflict)
- F (1 conflict)

Allocate in reverse order

- F, C, B, E, D, A

Register choice

- When allocating register for a variable
  - Make list of all candidates
  - Reject those that are already allocated to variable's enemies
    - enemy – variable of the same type that lives at the same time
  - From those left, take one that most friends are allocated to
    - friend – variable that we want to share allocation with
      - if we have “mov A, B” somewhere, A and B are friends
  - In case of ties, take a register with least pressure
    - Makes instruction scheduling easier

What about SSA ?

- Conversion to SSA introduces many new variables
- Conversion out of SSA introduces many MOVs
- Usually, numbered versions of one original variable
  - are not enemies – their lifespans are initially disjoint
  - are friends – because of the introduced MOVs
- So the final code often looks like SSA never happened
- Result different if actual optimizations happened
  - Extra MOVs more than compensated elsewhere

Application Binary Interface

- Low-level interface between parts of the system
  - Calling convention
    - How to pass arguments to a function
    - Where is return value
    - Which registers are preserved
  - Object layout
    - How to get normals from hit data
  - Register use patterns
- Hardware architecture does not fully determine ABI

Application Binary Interface

- Registers:
  - R15.x - R15.w are 4 scalar junk registers
  - Registers R0.w to R14.w are 15 scalar registers
  - Registers R0.xyz to R14.xyz are 15 vector/color registers
- Strings represented by hashes – first 24 bits of MD5
  - “world” → 8223024.0
  - 600 strings give collision probability ~1%, 4800 → ~50%
- Matrices not supported

Calling convention

- Function arguments passed in registers with smallest numbers, return value in R0
- Caller responsible for saving all registers
  - “Almost-assembler” CALL opcode does the saving
- color f(vector a, b; float c) {…}
  - a – R0.xyz, b – R1.xyz, c – R0.w, return value – R0.xyz

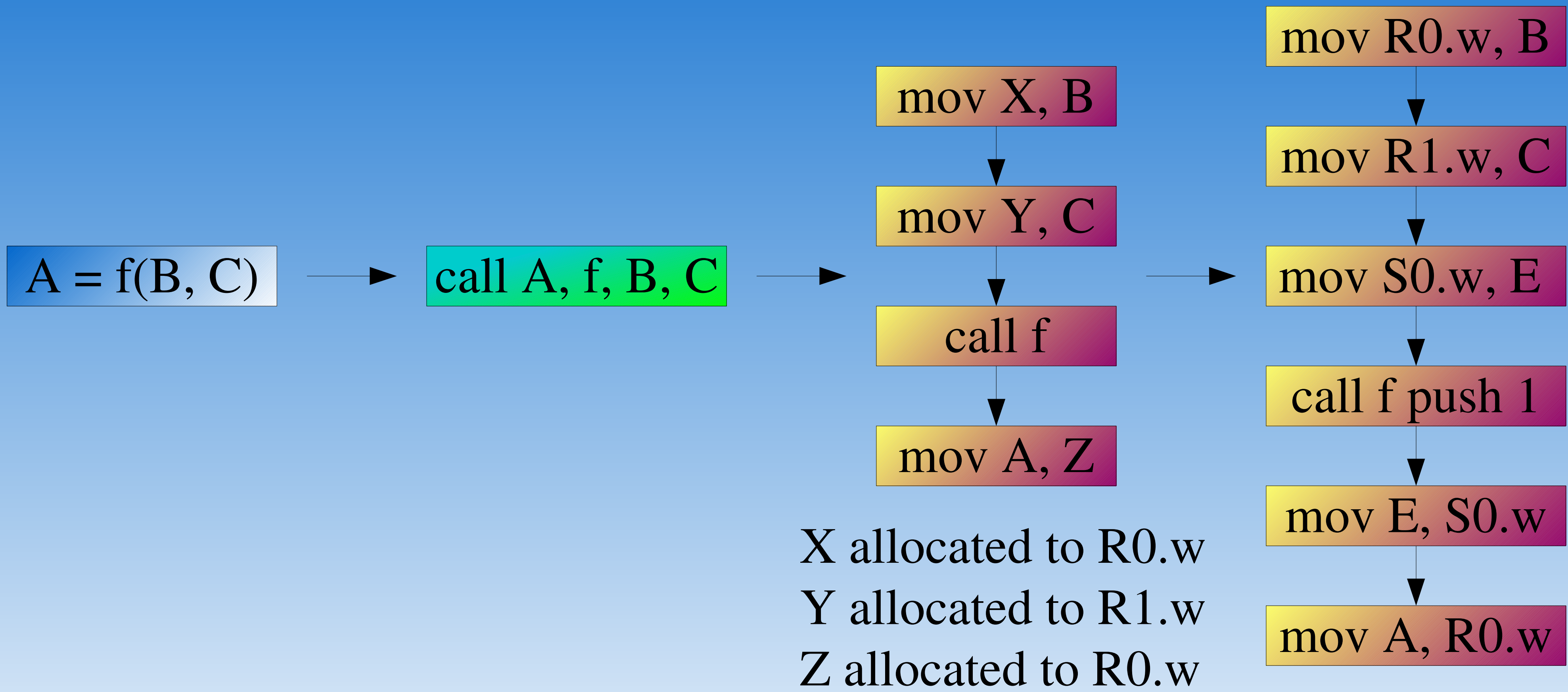
ABI and register allocation

- Do not hard-allocate variables, copy
- “f(float a)” generates:
  - X allocated to R0.w
  - mov a, X
- Friends allocation removes the MOV in most cases
- Hard-allocation of a would make it conflict with arguments to functions called from within f

CALL

- Argument passing, return values handled by variables hard-allocated to the right registers
- Registers are not preserved across function calls
- CALL opcode in pseudo-assembly pretends to preserve everything (except for S)
- At final code output, CALL uses its information which variables are live to selectively preserve some.

CALL


A = f(B, C)
call A, f, B, C
mov X, B
mov Y, C
call f
mov A, Z
X allocated to R0.w
Y allocated to R1.w
Z allocated to R0.w
mov R0.w, B
mov R1.w, C
mov S0.w, E
call f push 1
mov E, S0.w
mov A, R0.w

Function inlining

- Function calls are expensive
  - Small non-recursive functions are much faster if inlined
- Most standard library functions get inlined
  - In fact all of them for now
- No inlining for user-defined functions
- Late inlining (at code generation)
  - Early inlining would make more optimizations possible

Function inlining

Early inlining

Parse tree → SSA

B = length(A)

dp3 X, A, A

rsq Y, X

rcp B, Y

Late inlining

SSA → Assembly

call B, length, A

dp3 X, A, A

mov_rsq R15.w, X

mov_rcp R15.w, S.w

mov B, S.w

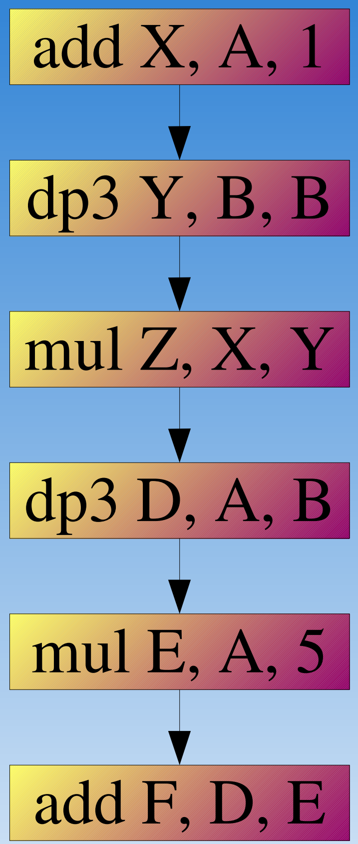
Instruction scheduling

- Runs after register allocation
  - “Good” scheduling increases register pressure
    - Can easily make code uncompilable as we run out of registers
  - “Good” register allocation doesn't makes scheduling harder
    - The worst case is minor performance degradation
  - Avoiding overcrowded registers makes scheduling easier
    - Other common register allocation strategy of using as few registers as possible makes scheduling extremely difficult
- NP-Hard, “List scheduling” algorithm used

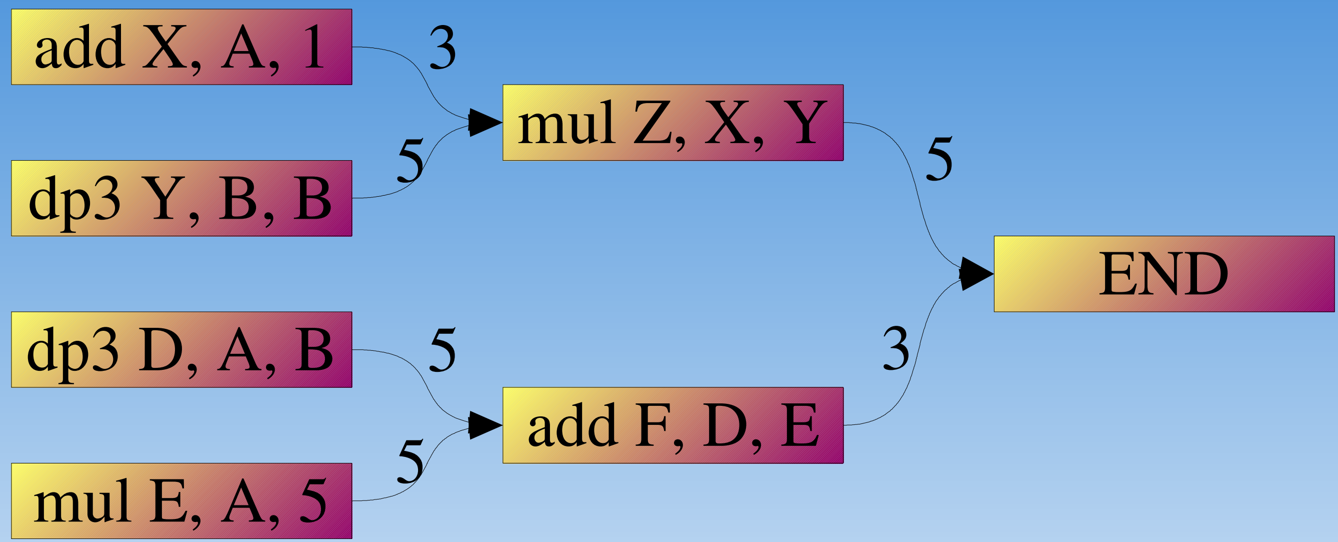
Instruction scheduling

- Divide function into basic blocks
  - Basic block – a series of instructions without branches
- For each block, compute dependency+latency graph
  - A depends on B if A cannot start until B finishes
- Until every instruction scheduled:
  - Possible to execute any instruction in this cycle ?
    - If so, execute one, if multiple candidates, select one by heuristics
    - If not, wait one cycle and retry

Instruction scheduling



If add has latency 3 and mul/dp3 – 5, original schedule takes 16 cycles



Instruction scheduling

1	dp3 Y, B, B	10, 5		
2	dp3 D, A, B	8, 5		
3	mul E, A, 5	8, 5		
4	add X, A, 1	8, 3		
5				
6			Y available	
7	mul Z, X, Y	5, 5	D available	X available
8	add F, D, E	3, 3	E available	
9				
10				
11			F available	
12	END		Z available	

Execution time reduced from 16 to 11 cycles

Instruction scheduling

- Heuristics:
  - Most cycles to the end
  - Most dependent instructions
  - Longest instruction latency
- Problems:
  - Basic blocks too small – every conditional breaks a block
  - Register allocation can introduce false dependencies
  - Efficiently using all four S registers difficult

Compatibility

- Compatibility with actual hardware not tested
- Hardware is a moving target
- Front-end (parser) not hardware-dependent
- Virtual machine can easily adapt
- Code generator can easily adapt

Summary

- Shader compiler is necessary for RPU
- Subset of Renderman Shading Language is good for writing shaders for RPU
- The compiler can generate highly efficient code from high-level shader specifications

Questions ?