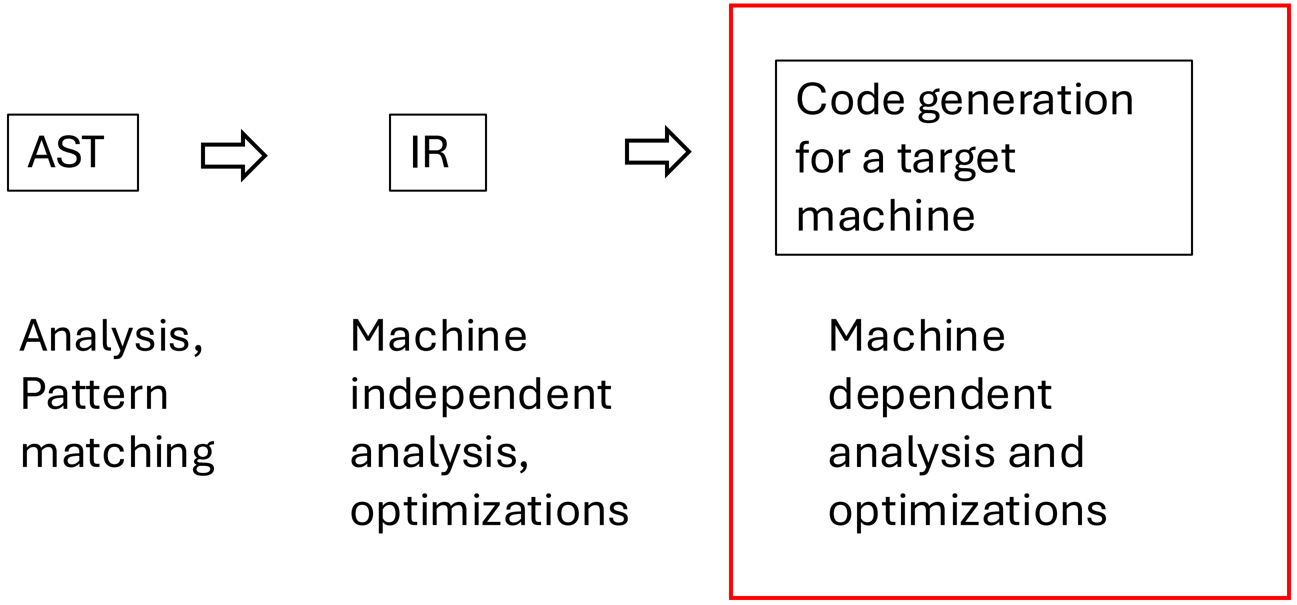
Code Generation – II
Scheduling, Instruction Selection

Nikhil Hegde

Compiler Optimizations in LLVM

Lecture series @ QUALCOMM Inc.

Compiler phases so far..


AST
IR
Code generation for a target machine
Analysis, Pattern matching
Machine independent analysis, optimizations
Machine dependent analysis and optimizations

LLVM Code Generator Steps

- Instruction Selection
  - LLVM IR -> DAG
- Scheduling and Formation
  - determine a schedule for DAG nodes
- SSA-based Machine Code Optimization
  - e.g. peephole optimizations
- Register Allocation
  - Unlimited virtual registers to limited machine registers. Introduce spill code if required.
- Prolog/Epilog Code generation
  - Function call conventions, frame pointer elimination
- Late Machine Code Optimization
  - Spill code scheduling, peephole optimizations
- Code Emission
  - Assembler code or direct machine code

https://llvm.org/docs/CodeGenerator.html

Recap: Instruction Selection

1. Build Initial DAG
2. Optimize Initial DAG
3. Legalize SelectionDAG Types
4. Optimize Selection DAG
5. Legalize SelectionDAG Operations
6. Optimize SelectionDAG
7. Select Instructions from SelectionDAG (also called DAG to DAG)
8. Selection DAG scheduling and Formation

Last class: What each step does

This class: how are 7 and 8 done

Recap: Legalize Type

```
1.   Optimized lowered selection DAG: %bb.0 'foo:'
2.   SelectionDAG has 15 nodes:
3.       t0: ch,glue = EntryToken
4.       t2: i32,ch = CopyFromReg t0, Register:i32 %0
5.                   t3: i16 = truncate t2
6.               t5: i16 = add t3, Constant:i16<5>
7.           t8: i1 = setcc t5, Constant:i16<6>, setugt:ch
8.       t10: i32 = select t8, t2, Constant:i32<9>
9.       t13: ch,glue = CopyToReg t0, Register:i32 $eax, t10
10.      t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 $eax, t13:1
```

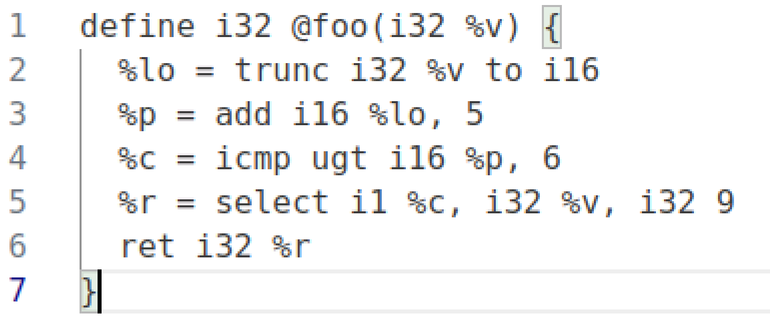
```
1.   Type-legalized selection DAG: %bb.0 'foo:'
2.   SelectionDAG has 17 nodes:
3.       t0: ch,glue = EntryToken
4.       t2: i32,ch = CopyFromReg t0, Register:i32 %0
5.                       t3: i16 = truncate t2
6.                   t5: i16 = add t3, Constant:i16<5>
7.               t15: i8 = setcc t5, Constant:i16<6>, setugt:ch
8.           t18: i8 = and t15, Constant:i8<1>
9.       t10: i32 = select t18, t2, Constant:i32<9>
10.      t13: ch,glue = CopyToReg t0, Register:i32 $eax, t10
11.      t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 $eax, t13:1
```

Recap: Legalize Type

- Promotion
- Expansion
- Soften
- Split vector
- Widen vector

Recap:Legalize Operation

```
Type-legalized selection DAG: %bb.0 'foo:'
SelectionDAG has 17 nodes:
  t0: ch,glue = EntryToken
  t2: i32,ch = CopyFromReg t0, Register:i32 %0
                  t3: i16 = truncate t2
                t5: i16 = add t3, Constant:i16<5>
              t15: i8 = setcc t5, Constant:i16<6>, setugt:ch
            t18: i8 = and t15, Constant:i8<1>
          t10: i32 = select t18, t2, Constant:i32<9>
  t13: ch,glue = CopyToReg t0, Register:i32 $eax, t10
  t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 $eax, t13:1
```



```
define i32 @foo(i32 %v) {
  %lo = trunc i32 %v to i16
  %p = add i16 %lo, 5
  %c = icmp ugt i16 %p, 6
  %r = select i1 %c, i32 %v, i32 9
  ret i32 %r
}
```

```
Legalized selection DAG: %bb.0 'foo:'
SelectionDAG has 15 nodes:
  t0: ch,glue = EntryToken
  t2: i32,ch = CopyFromReg t0, Register:i32 %0
                  t3: i16 = truncate t2
                t5: i16 = add t3, Constant:i16<5>
              t20: i16,i32 = X86ISD::SUB t5, Constant:i16<7>
            t23: i32 = X86ISD::CMOV Constant:i32<9>, t2, TargetConstant:i8<3>, t20:1
  t13: ch,glue = CopyToReg t0, Register:i32 $eax, t23
  t14: ch = X86ISD::RET_GLUE t13, TargetConstant:i32<0>, Register:i32 $eax, t13:1
```

Recap: Legalize Operation

- Expansion
- LibCall
- Promotion
- Custom

Instruction Scheduling

Instruction Scheduling - Considerations

- Gather constraints on schedule:
    - Data dependences between instructions
    - Resource constraints
- Schedule instructions while respecting constraints
    - List scheduling
    - Height-based heuristic

Representing constraints

- **Dependence** constraints and **resource** constraints limit valid orders of instructions
- Instruction scheduling goal:
  - For each instruction in a program (basic block), assign it a *scheduling slot*
  - Which functional unit to execute on, and when
  - As long as we obey all of the constraints
- So how do we represent constraints?

Data dependence graph

- Graph that captures data dependence constraints
- Each node represents one instruction
- Each edge represents a dependence from one instruction to another
- Label edges with instruction *latency* (how long the first instruction takes to complete → how long we have to wait before scheduling the second instruction)

Example

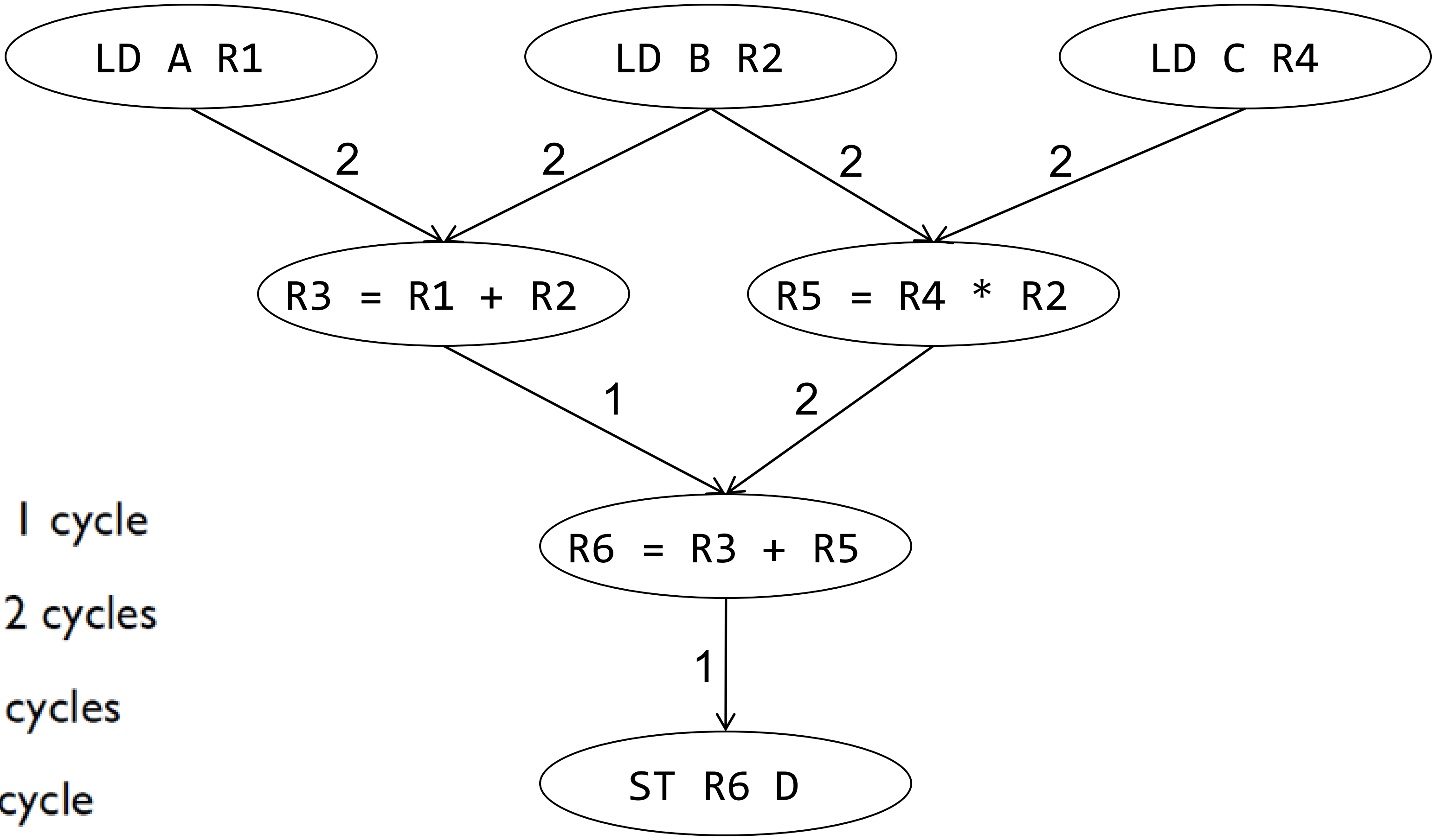
- ADD takes 1 cycle
- MUL takes 2 cycles
- LD takes 2 cycles
- ST takes 1 cycle

```
LD A, R1
LD B, R2
R3 = R1 + R2
LD C, R4
R5 = R4 * R2
R6 = R3 + R5
ST R6, D
```

Example

LD A, R1
LD B, R2
R3 = R1 + R2
LD C, R4
R5 = R4 * R2
R6 = R3 + R5
ST R6, D

- ADD takes 1 cycle
- MUL takes 2 cycles
- LD takes 2 cycles
- ST takes 1 cycle



Reservation tables

- Represent resource constraints using reservation tables
- For each instruction, table shows which functional units are occupied in each cycle the instruction executes
  - # rows: latency of instruction
  - # columns: number of functional units
  - T[i][j] marked ⇔ functional unit *j* occupied during cycle *i*
    - Caveat: some functional units are *pipelined*: instruction takes multiple cycles to complete, but only occupies the unit for the first cycle
- Some instructions have multiple ways they can execute: one table per variant

Example

- Two ALUs, fully pipelined
- One LD/ST unit, *not pipelined*
- ADDs can execute on ALU0 or ALU1
- MULs can execute on ALU0 only
- ADD takes 1 cycle
- MUL takes 2 cycles
- LD takes 2 cycles
- ST takes 1 cycle
- LOADs and STOREs both occupy the LD/ST unit

ALU0	ALU1	LD/ST

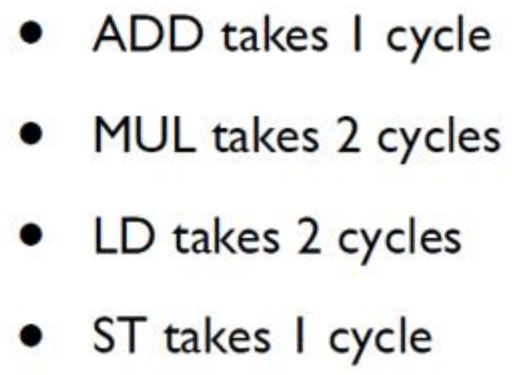
Example

- Two ALUs, fully pipelined
- One LD/ST unit, *not pipelined*

ALU0	ALU1	LD/ST

Example

- Two ALUs, fully pipelined
- One LD/ST unit, *not pipelined*
- ADDs can execute on ALU0 or ALU1



- ADD takes 1 cycle
- MUL takes 2 cycles
- LD takes 2 cycles
- ST takes 1 cycle

ALU0	ALU1	LD/ST
X		

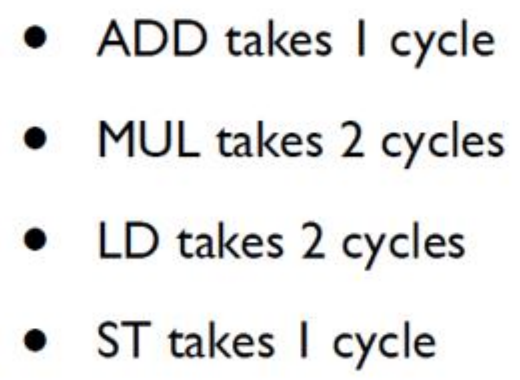
ADD (1)

ALU0	ALU1	LD/ST
	X	

ADD (2)

Example

- Two ALUs, fully pipelined
- One LD/ST unit, *not pipelined*
- ADDs can execute on ALU0 or ALU1
- MULs can execute on ALU0 only



- ADD takes 1 cycle
- MUL takes 2 cycles
- LD takes 2 cycles
- ST takes 1 cycle

ALU0	ALU1	LD/ST
X		

MUL

Example

- Two ALUs, fully pipelined
- One LD/ST unit, *not pipelined*
- ADDs can execute on ALU0 or ALU1
- MULs can execute on ALU0 only
- LOADs and STOREs can execute on LD/ST unit only

- ADD takes 1 cycle
- MUL takes 2 cycles
- LD takes 2 cycles
- ST takes 1 cycle

ALU0	ALU1	LD/ST
		X
		X

LOAD

ALU0	ALU1	LD/ST
		X

STORE

Example

ADD(1)

ALU0	ALU1	LD/ST
X		

LOAD

ALU0	ALU1	LD/ST
		X
		X

ADD(2)

ALU0	ALU1	LD/ST
	X	

STORE

ALU0	ALU1	LD/ST
		X

MUL

ALU0	ALU1	LD/ST
X		

MUL still takes two cycles. Since ALU is fully pipelined, only occupies the ALU for 1

Can use reservation tables to see if instructions can be scheduled: see if tables overlap

Using tables

ADD(1)

ALU0	ALU1	LD/ST
X		

LOAD

ALU0	ALU1	LD/ST
		X
		X

ADD(2)

ALU0	ALU1	LD/ST
	X	

STORE

ALU0	ALU1	LD/ST
		X

MUL

ALU0	ALU1	LD/ST
X		

Which of the sequences below are valid?
| = run instructions in same cycle
; = move to next cycle

ADD | ADD	✓
ADD | MUL	✓
MUL | MUL	✗
MUL ; MUL | ADD	✓
LOAD | MUL	✓
LOAD ; STORE	✗
STORE ; LOAD	✓

Scheduling

- Can use these constraints to schedule a program
- Data dependence graph tells us what instructions are *available for scheduling* (have all of their dependences satisfied)
- Reservation tables help us build schedule by telling us which functional units are occupied in which cycle

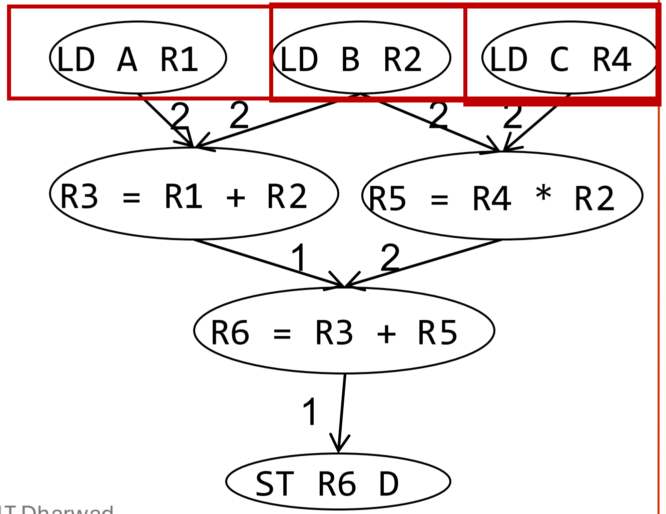
List scheduling

1. Start in cycle 0
2. For each cycle
   1. Determine which instructions are available to execute
   2. From list of instructions, pick one to schedule, and place in schedule
   3. If no more instructions can be scheduled, move to next cycle

Cycle	ALU0	ALU1	LD/ST
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

List scheduling - Example

```
1. LD A, R1
2. LD B, R2
3. R3 = R1 + R2
4. LD C, R4
5. R5 = R4 * R2
6. R6 = R3 + R5
7. ST R6, D
```



Cycle #	Available Instruction(s)	Scheduled Instruction(s)	Completed Instruction(s)
0	1, 2, 4	1*	
1	2, 4		
2	2, 4	2*	1
3	4		
4	3,4	3,4	2
5			3
6	5	5	4
7			
8	6	6	5
9	7	7	6
10			7

*an instruction from the list of available instructions is picked at random and scheduled

List scheduling

1. LD A, R1
2. LD B, R2
3. R3 = R1 + R2
4. LD C, R4
5. R5 = R4 * R2
6. R6 = R3 + R5
7. ST R6, D

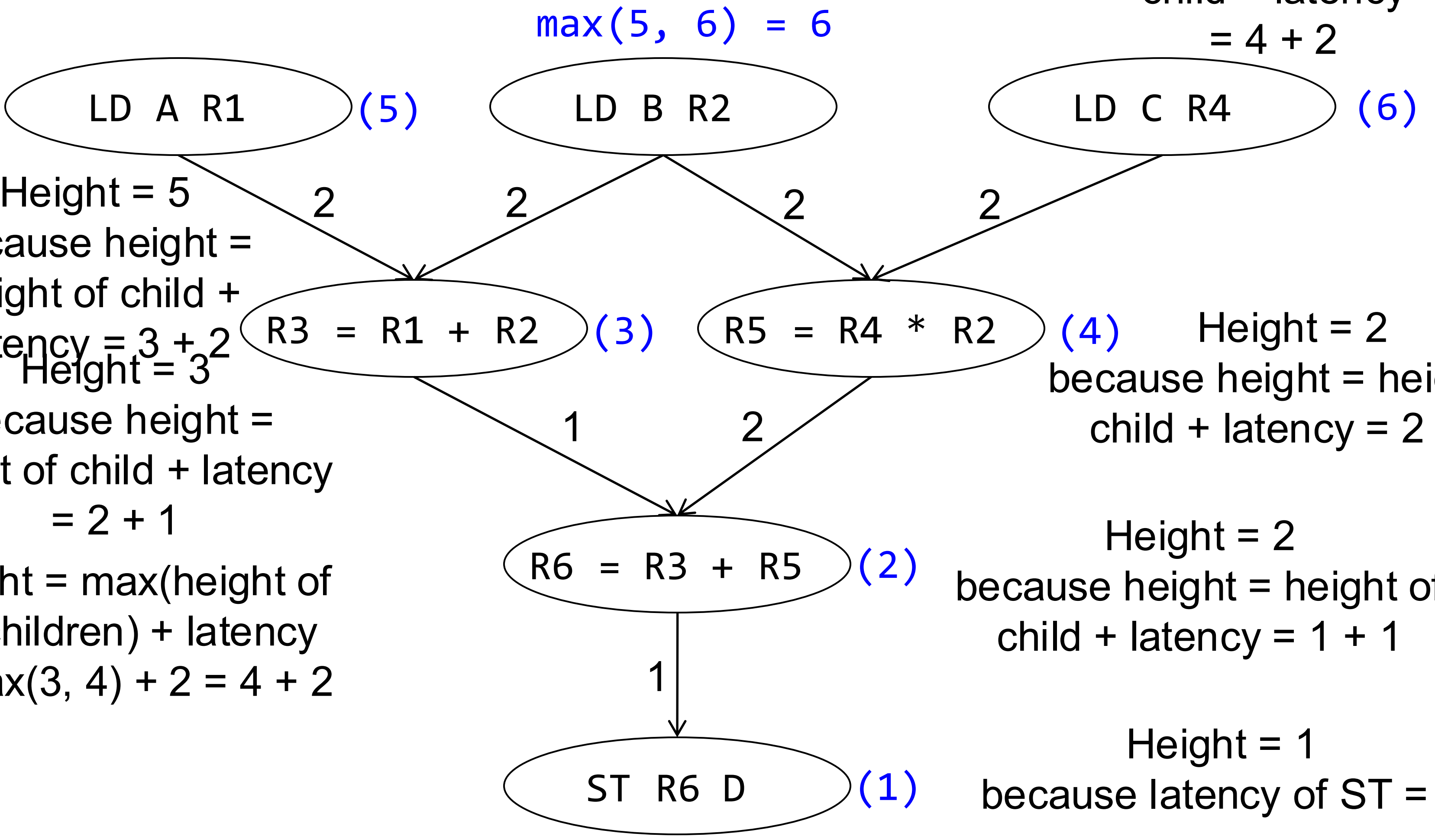
Cycle	ALU0	ALU1	LD/ST
0			1
1			1
2			2
3			2
4	3		4
5			4
6	5		
7			
8	6		
9			7
10			

Height-based scheduling

- Important to prioritize instructions
  - Instructions that have a lot of downstream instructions dependent on them should be scheduled earlier
- Instruction scheduling NP-hard in general, but **height-based scheduling** is effective
- Instruction height = latency from instruction to farthest-away leaf
  - Leaf node height = instruction latency
  - Interior node height = max(heights of children + instruction latency)
- Schedule instructions with highest height first

Computing heights

Height = height of child + latency = 4 + 2



Height = 5 because height = height of child + latency = 3 + 2

Height = 3 because height = height of child + latency = 2 + 1

Height = max(height of all children) + latency = max(3, 4) + 2 = 4 + 2

Height = 2 because height = height of child + latency = 2 + 2

Height = 2 because height = height of child + latency = 1 + 1

Height = 1 because latency of ST = 1

Height-based list scheduling

1. LD A, R1
2. LD B, R2
3. R3 = R1 + R2
4. LD C, R4
5. R5 = R4 * R2
6. R6 = R3 + R5
7. ST R6, D

Cycle	ALU0	ALU1	LD/ST
0			2
1			2
2			4
3			4
4	5		1
5			1
6	3		
7	6		
8	7		
9			
10			

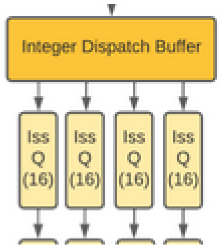
Specifying Resource Constraints in LLVM

```
// From llvm/lib/Target/RISCV/RISCVInstrInfoM.td.
def DIV : ALU_rr<0b0000001, 0b100, "div">,  Sched<[WriteIDiv, ReadIDiv, ReadIDiv]>;

// Integer division def : WriteRes<WriteIDiv, [SiFive7PipeB, SiFive7IDiv]> {
  let Latency = 66;
  let ReleaseAtCycles = [1, 65];
}

def SiFiveP600IEXQ0 : ProcResource<1>;
def SiFiveP600IEXQ1 : ProcResource<1>;
def SiFiveP600IEXQ2 : ProcResource<1>;
def SiFiveP600IEXQ3 : ProcResource<1>;
...
def SiFiveP600IntArith : ProcResGroup<[SiFiveP600IEXQ0, SiFiveP600IEXQ1,
SiFiveP600IEXQ2, SiFiveP600IEXQ3]>;

// Integer arithmetic and logic def : WriteRes<WriteIALU,
[SiFiveP600IntArith]>;
```



https://myhsu.xyz/llvm-sched-model-1/

Specifying Resource Constraints in LLVM

- Additional specifications

```
// Only one Branch unit. def BR :
ProcResource<1> {
   let BufferSize = 16;
 }
```

```
def SiFiveP600Model : SchedMachineModel {
   let IssueWidth = 4; // 4 micro-ops are dispatched per cycle.
   let MicroOpBufferSize = 160; // Max micro-ops that can be buffered.
   ...
}
```

Excellent reference: https://myhsu.xyz/llvm-sched-model-1.5/

From Optimize2 to Instruction Selection in SelectionDAG: Tablegen files

```
// from: X86RegisterInfo.td
//Define a class for 32-bit floating-point registers (XMM)
def XMM0 : X86Reg<0, "xmm0">;
def XMM1 : X86Reg<1, "xmm1">;
...
def XMM31 : X86Reg<31, "xmm31">;
// Register class grouping all 32-bit float registers
def FR32 : RegisterClass<"X86", [f32], 32, (sequence "XMM%u", 0, 31)>;
```

```
//from: X86InstrSSE.td
def DIVSSrr :
 SDI<0x5E, MRMSrcReg, (outs FR32:$dst), (ins FR32:$src1, FR32:$src2),
   "divss\t{$src2, $dst|$dst, $src2}",
   [(set FR32:$dst, (fdiv FR32:$src1, FR32:$src2))]>;
```

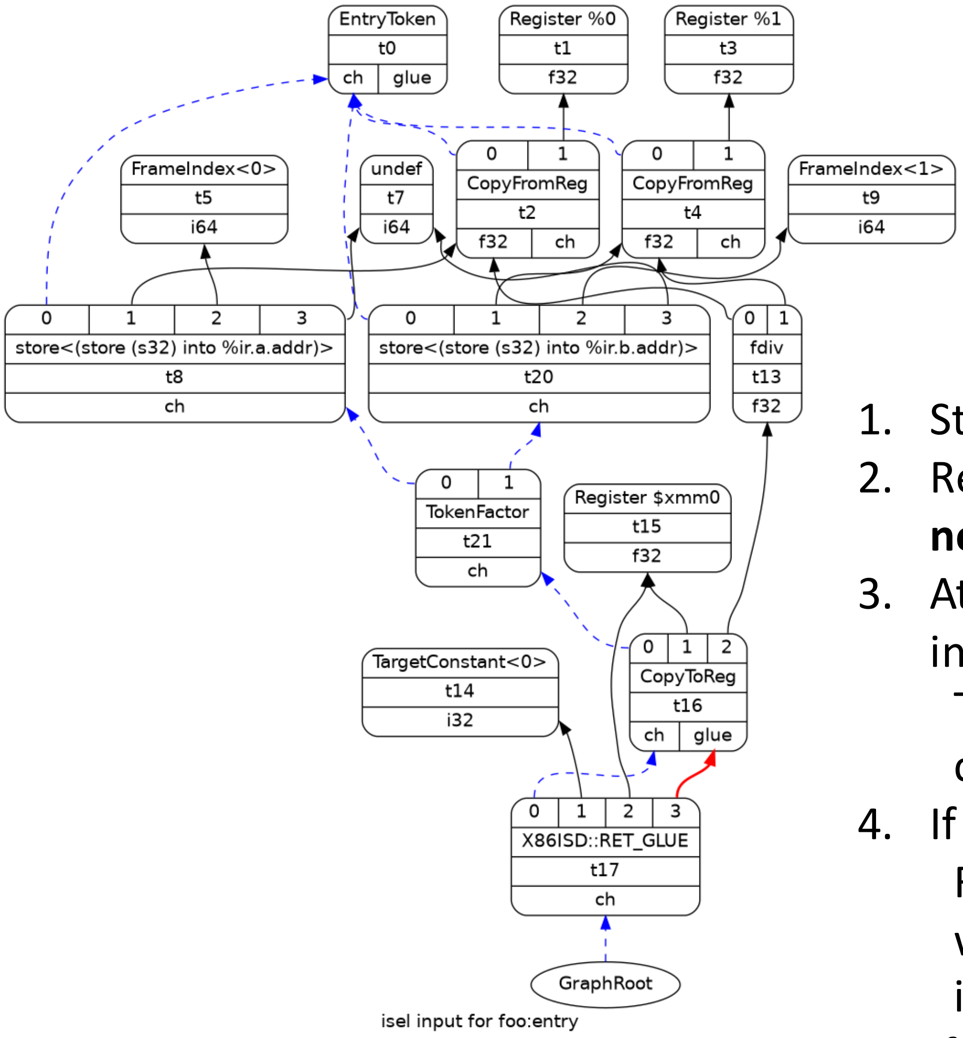
---

t13: f32 = fdiv t2, t4 ➡ t13: f32 = DIVSSrr nofpexcept t2, t4 ➡ divss %xmm1, %xmm0

C++ code generation

- set FR32:$dst, (fdiv FR32:$src1, FR32:$src2))

```
if (N->getOpcode() == ISD::FDIV && N->getValueType(0) ==
MVT::f32) {
  SDValue Op0 = N->getOperand(0);
  SDValue Op1 = N->getOperand(1);
  if (isRegisterClass(Op0, FR32) && isRegisterClass(Op1, FR32)){
   ReplaceNode(N, CurDAG->getMachineNode(X86::DIVSSrr, DL,
MVT::f32, Op0, Op1));
   return;
  }
}
```



DAG Traversal for performing Step 7

1. Start at the **root node** (RET_GLUE).
2. Recursively visit all operands until reaching **leaf nodes** (constants, frame indices, registers, etc.).
3. At each node, attempt to match patterns defined in TableGen:
   The matcher tests if the current node and its children match any instruction's pattern.
4. If a match is found:
   Replace the node (and possibly its subgraph) with one corresponding to the target instruction.
5. If no pattern matches:
   Use fallback selection to expand into simpler operations or lower it into libCall sequences.
6. Continue bottom-up until the root is fully selected.

Using Tablegen for Specifying..

- Instructions
- Operands
- Registers
- Calling conventions
- Scheduler models
- Target-specific features
- Intrinsics
- IR patterns for pattern matching (e.g., in SelectionDAG)

Motivation for GlobalISel

- Performance
  - Avoid DAG
- Operate at Function Granularity
- Modular