Scheduler for in-order processors

What's present and what's missing in LLVM?

ARM

Javed Absar

Staff Engineer, Compilation Tools

LLVM Cauldron, Hebden-Bridge
September 8, 2016

Contents

- Introduction to LLVM Machine Scheduler
- Carp - A generic in-order processor
- Develop Scheduler for Carp in LLVM
- Discussion – what's present and what's missing?
- Conclusion

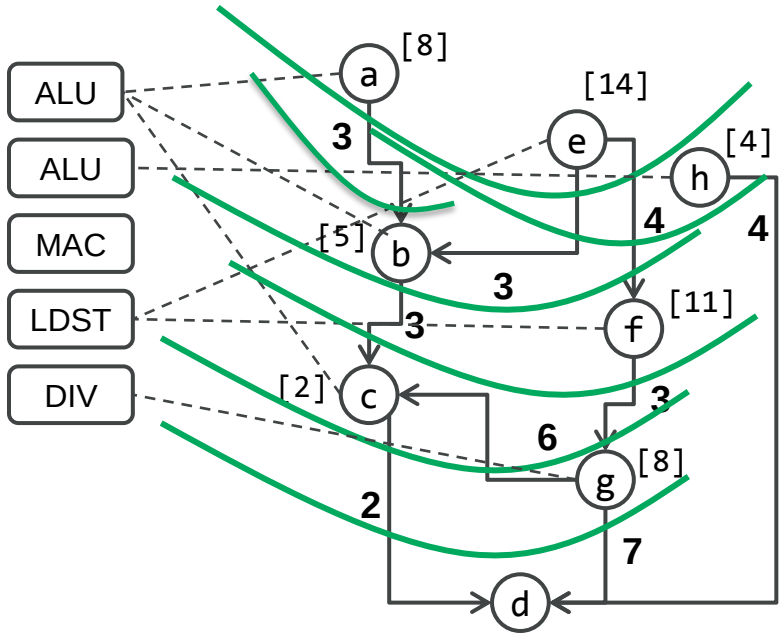
Progress

Instruction Scheduling

- Re-order instructions to improve performance
  - Reduce pipeline stalls, increase IPC
- In-order vs Out-of-order
  - In-order: fetched, executed & complete in compiler generated program order
    - Statically scheduled
  - Out-of-order: fetch and completion in order
    - Lockup-free cache, speculative execution
    - Dynamically scheduled
- Focus on in-order processor in this presentation

Instruction Scheduling

- General scheduling algorithms -
  - Create dependence graph
  - Topological sort– many choices NPC
  - List scheduling (heuristic, sub-optimal solutions)
    - Decreasing time, critical path, register pressure, clustering,
- Require –
  - latencies, resource cycles, micro-ops



ARM

LLVM Scheduler

- Machine scheduler (MIScheduler) ~ 2012
  - Per-operand approach
  - Latest and greatest, adapted as Scheduler of the future
  - List Scheduler (with prioritization)
  - Function Pass, works at Machine Instrs level (ScheduleDAG/SUnit)

- Previously, in LLVM
  - Before ~ 2008: At end of SelectionDAG
  - SDNodes (ScheduleDAG)– scheduling units

LLVM Machine Scheduler

- Enabling and using –

```
$ clang ...-mllvm -enable-misched -mllvm -enable-post-misched -mllvm -misched-postra
```

```
$ llc ... -mcpu=carp  -enable-misched  -enable-post-misched -misched-postra -misched-topdown
```

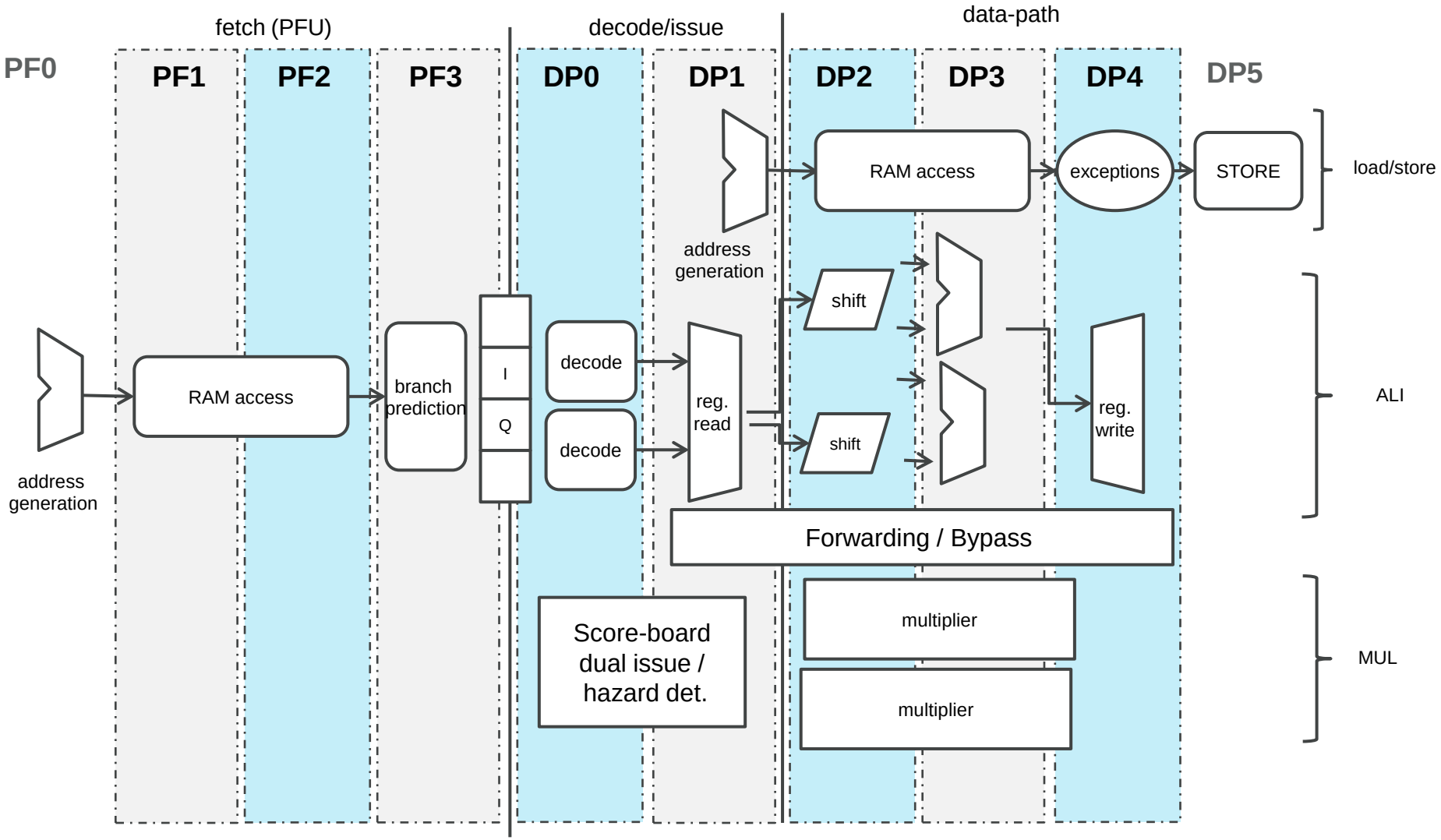
- Choose policy
  - -misched-topdown, -misched-bottom-up
- Choose options
  - –misched-cutoff –misched-limit –misched-regpressure –misched-cyclicpath –misched-cluster

Progress

CARP – A generic in-order processor

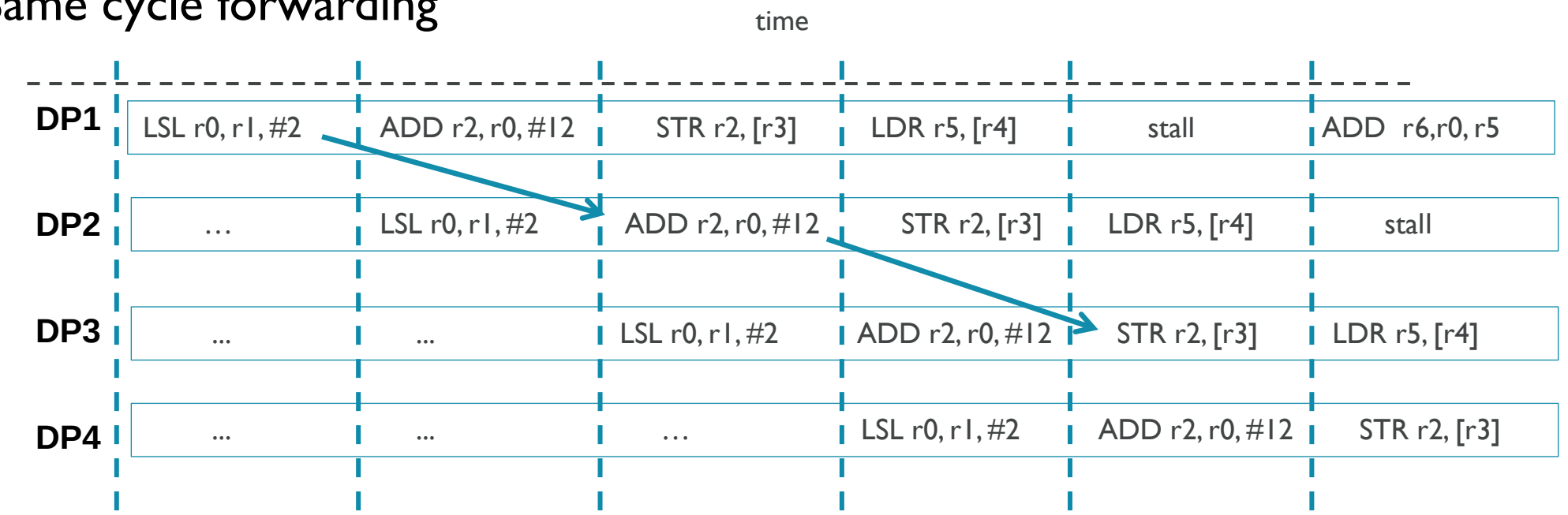
- Functional Units
  - Fully Pipelined, forwarding facility
  - 2 ALU, 2 MUL, 1 DIV, 1 LD/ST
- Instruction fetch
  - Fetch multiple instructions data per cycle
  - Instruction Queue (IQ)
    - De-couple instruction fetch from decode/execution
- Multiple issue
  - Issue up to two instructions (in program order)
  - Once issued proceed together
- Score-board
  - Issues instructions when data, resource available
  - Status tables
- Branch Prediction
  - Predict call-return address, branch-taken
  - Pipeline flush in case of mis-prediction

CARP Pipeline



CARP Pipeline - Forwarding

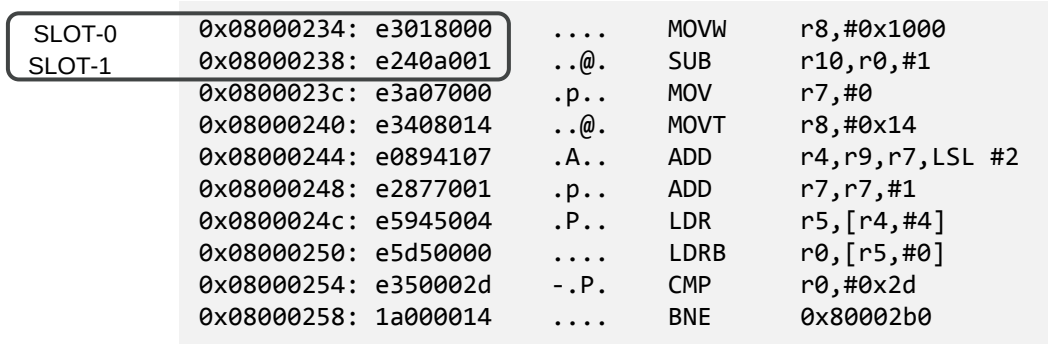
- Consumed - when value is consumed
- Produced– earliest when value is produced
- Complex map of different 'Produced x Consumed'
- Same cycle forwarding



ARM

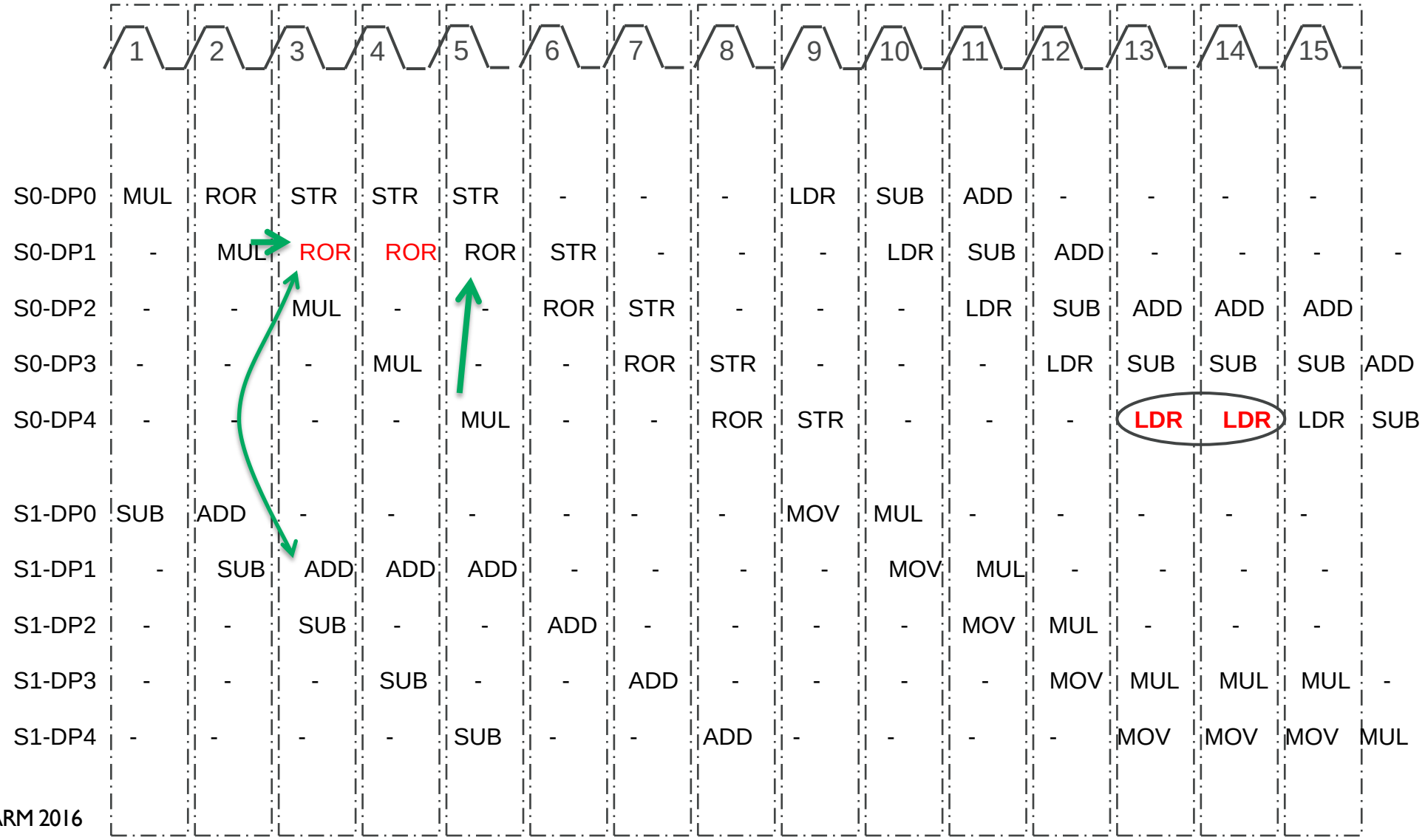
CARP Pipeline – Dual Issue

- Earlier (or sole in case of single issue) instruction - slot-0
- Sliding window
- Cannot independently interlock

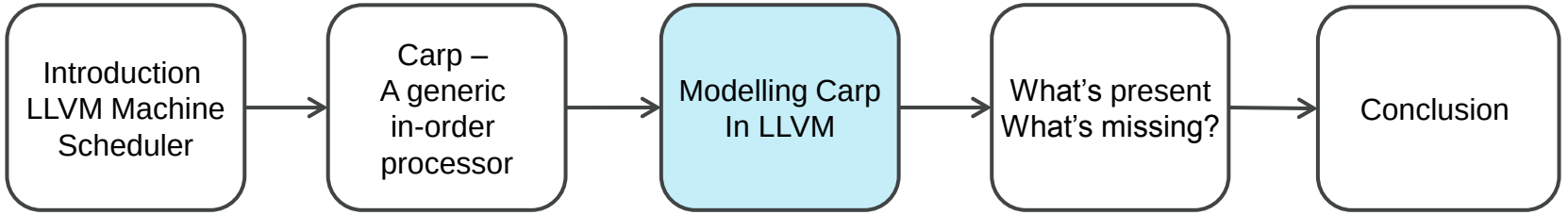


- Not dual-issued if:
  - Structural dependence or data dependence
  - slot-1 instruction increases interlock
  - Multi-cycle instructions (dual issue only in last cycle)
  - Loads to PC

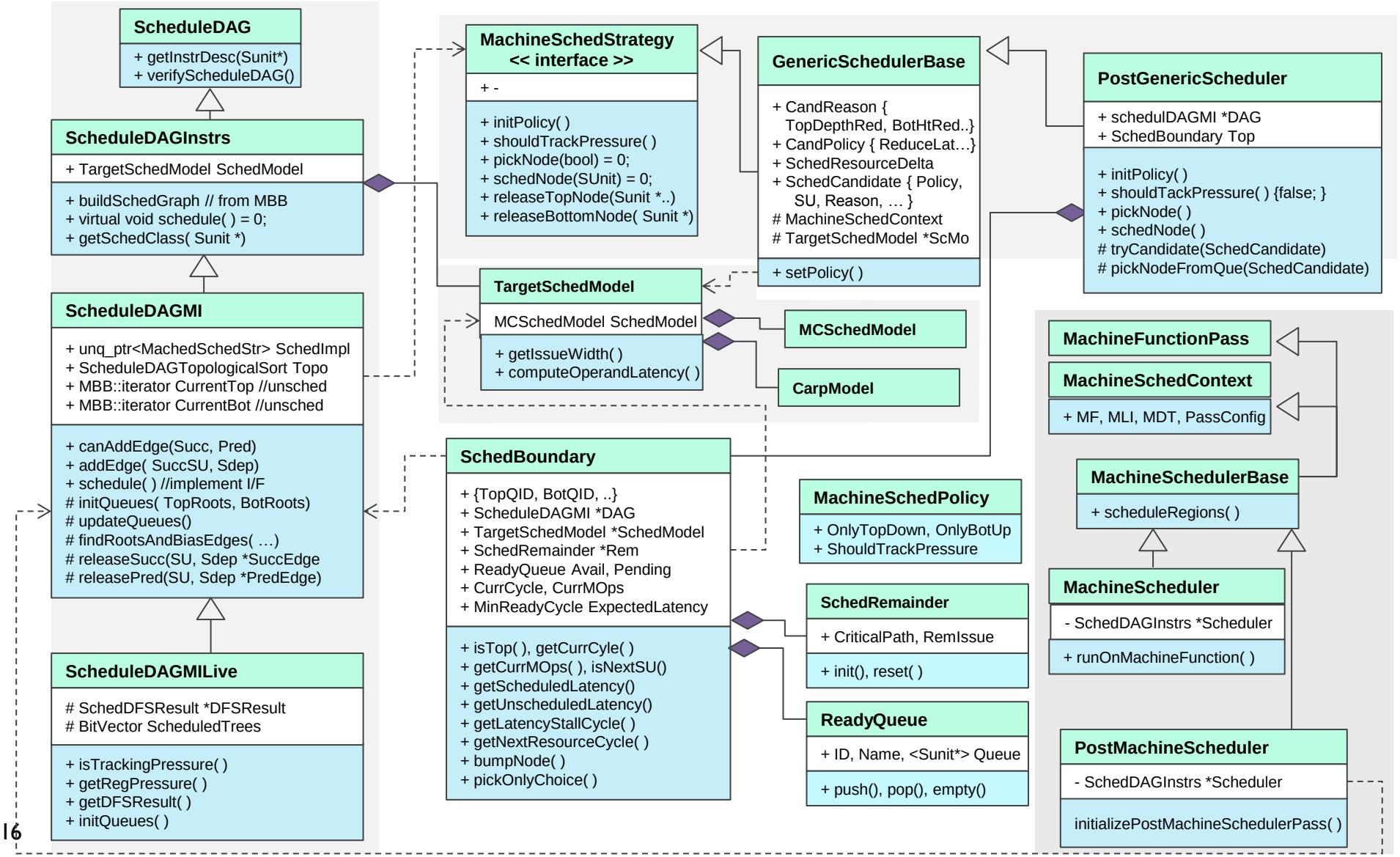
Instruction Flow in pipeline



Progress



Machine Scheduler in LLVM



CARP Sched Machine Model

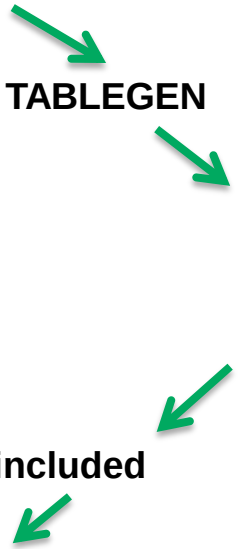
- Top-level pipeline description

```
// ===---------------------------------------------------------------------===//
// The following definitions describe the per-operand machine model.
// This works with MachineScheduler. See TargetSchedule.td for details.

def CarpModel : SchedMachineModel {
  let MicroOpBufferSize = 0;  // Carp is in-order processor
  let IssueWidth = 2;         // 2 micro-ops dispatched per cycle
  let LoadLatency = 1;        // Optimistic, assuming no misses
  let MispredictPenalty = 8;  // A branch direction mispredict, including PFU
  let PostRAScheduler = 1;    // Enable PostRA scheduler pass.
  let CompleteModel = 0;      // FIXME: Remove if all instructions are covered.
}
```

LLVM Scheduler

```
//==- FishScheduleCarp.td - Carp Scheduling Definitions -*- tablegen -*-=//
def CarpModel : SchedMachineModel {
  let MicroOpBufferSize = 0;  // Carp is in-order processor
  let IssueWidth = 2;         // 2 micro-ops dispatched per cycle
  let LoadLatency = 1;        // Optimistic, assuming no misses
  let MispredictPenalty = 8;  // A branch direction mispredict, including PFU
  let PostRAScheduler = 1;    // Enable PostRA scheduler pass.
  let CompleteModel = 0;      // FIXME: Remove if all instructions are covered.
}
…
```




```
//==- FishGenSubtargetInfo.inc ----------------------------------*/
/*===- TableGen'erated file ---------------------*- C++ -*-===*\
|* Automatically generated file, do not edit!                   *|..*/
…
static const llvm::MCSchedModel CarpModel = {
  2, // IssueWidth
  0, // MicroOpBufferSize
  MCSchedModel::DefaultLoopMicroOpBufferSize,
  1, // LoadLatency
   …
  CarpModelProcResources,
  CarpModelSchedClasses,
};
…
extern const llvm::MCWriteLatencyEntry FishWriteLatencyTable[] = { …
}
```

```
//===-- lib/Target/Fish/FishMCTargetDesc.cpp - Fish Target Descriptions ---------------------===//
…
#define GET_SUBTARGETINFO_MC_DESC
#include “FishGenSubtargetInfo.inc"
```

CARP Sched Machine Model

- Target general approach
  - Step 1 - Modify instruction definition to inherit from Sched
    - Associate each operand with SchedReadWrite type (input/output operand)
  - Step 2 – In sub-target, WriteRes and ReadAdvance
    - Associate resource and latency for each SchedReadWrite type
- Sub-target specific approach
  - Define own SchedReadWrite types
  - Associate resources / latencies through SchedWriteRes
  - Associate forwards through SchedReadAdvance
  - Map SchedReadWrite types to specific opcodes/operands

CARP Sched Machine Model

```
// FishSchedule.td
def WriteALU : SchedWrite;
```

```
// FishInstrInfo.td
class ALUInst<bits<4> Opc, string OpcodeStr, SDNode OpNode>
 : InstFish<(outs GPR:$dst), (ins GPR:$src2, GPR:src1),
      !stroncat(OpcdeStr, "\t$dst, $src2, $src1"),
      [(set GPR:$dst, (OpNode GPR:$src2, GPR:$src1))]>, Sched<[WriteALU]> {…}
```

```
//===----------------------------------------------------------------------===//
// Subtarget-specific SchedWrite types which both map the ProcResources and
// set the latency.
let SchedModel = CarpModel in {
// ALU - Write occurs in Late EX2 (independent of whether shift was required)
def : WriteRes<WriteALU, [CarpUnitALU]> { let Latency = 3; }
def : WriteRes<WriteALUsi, [CarpUnitALU]> { let Latency = 4; }
def : WriteRes<WriteALUsr, [CarpUnitALU]> { let Latency = 4; }
def : WriteRes<WriteALUSsr, [CarpUnitALU]> { let Latency = 4; }

// Loads
def : WriteRes<WriteLd, [CarpUnitLd]> { let Latency = 4; } // WRI
…
}
```

CARP Sched Machine Model

- Sub-target specific SchedWrite
  - Associate resources with SchedWrites types

```
// Carp specific SchedWrites for use with InstRW
def CarpWriteMAC      : SchedWriteRes<[CarpUnitMAC]> { let Latency = 4; }
def CarpWriteLd       : SchedWriteRes<[CarpUnitLd]> { let Latency = 4; }
...
def CarpWriteALU_DP2  : SchedWriteRes<[CarpUnitALU]> { let Latency = 2; }
def CarpWriteALU_DP3  : SchedWriteRes<[CarpUnitALU]> { let Latency = 3; }

...
// FP division takes fixed #cycles
def CarpWriteFPDIV_SP : SchedWriteRes<[CarpUnitFPDIV]> { let Latency = 7; }
def CarpWriteFPDIV_DP : SchedWriteRes<[CarpUnitFPDIV]> { let Latency = 17; }
...
```

```
// TargetSchedule.td
class SchedWriteRes<list<ProcResourceKind> resources> : SchedWrite,
ProcWriteResources<resources>;
```

CARP Sched Machine Model

- ReadAdvance –
  - Forwarding/ByPass information for SchedReads

```
// Carp specific SchedReads - related to pipeline stages
def CarpRead_ISS : SchedRead;
def CarpRead_EX1 : SchedRead;
def CarpRead_EX2 : SchedRead;
def CarpRead_WRI : SchedRead;
def CarpRead_F0  : SchedRead; // FP-F0 maps to DP1 stage of integer pipe
def CarpRead_F1  : SchedRead;
def CarpRead_F2  : SchedRead;
…
// Forwarding information - based on when an operand is read
def : ReadAdvance<CarpRead_DP1, 0>;
def : ReadAdvance<CarpRead_DP2, 1>;
def : ReadAdvance<CarpRead_DP3, 2>;
def : ReadAdvance<CarpRead_F0, 0>;
def : ReadAdvance<CarpRead_F1, 1>;
def : ReadAdvance<CarpRead_F2, 2>;
```

CARP Sched Machine Model

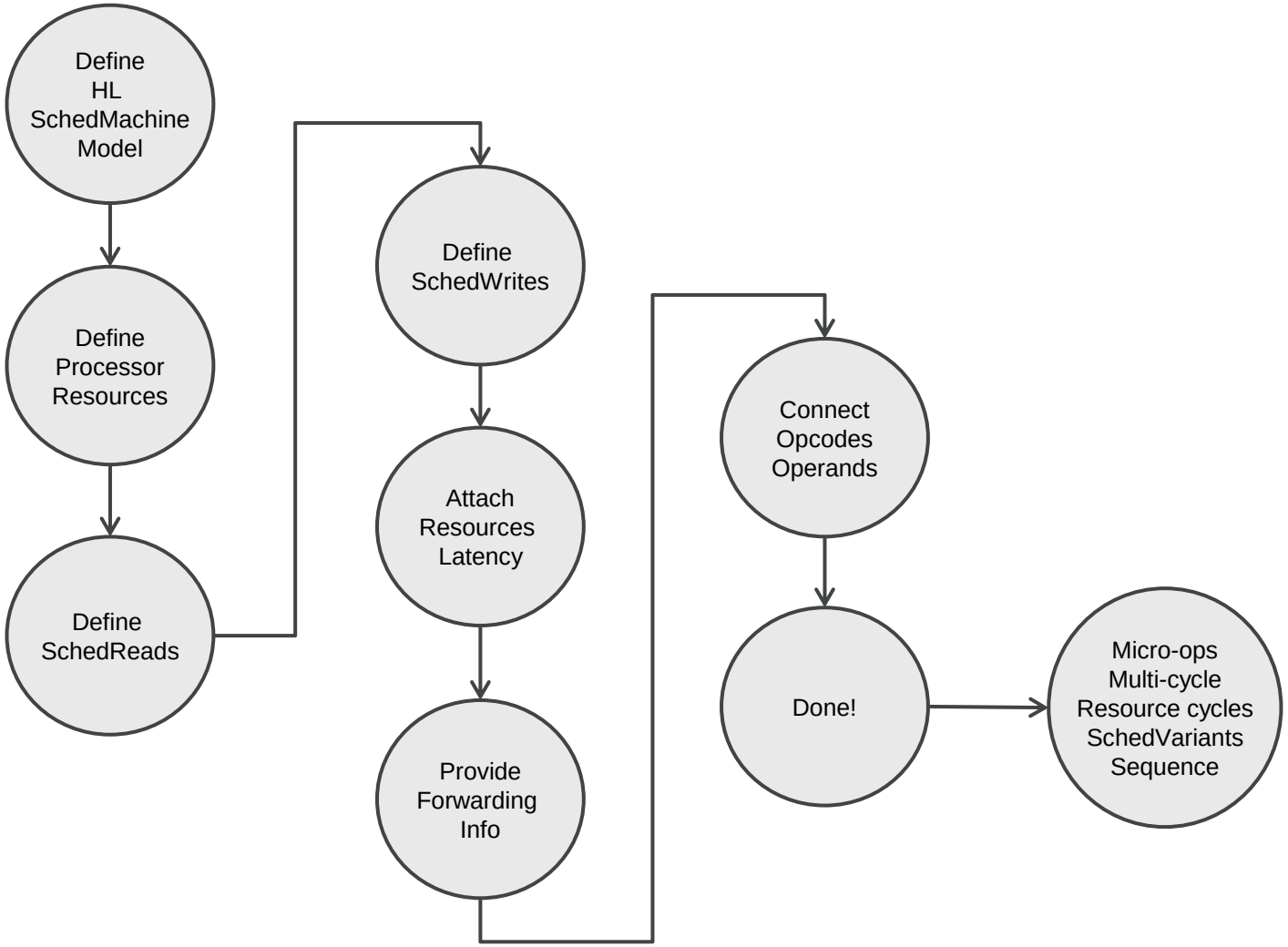
- Sub-target specific SchedWrites

```
def : InstRW<[CarpWriteALU_DP3, CarpRead_DP2], (instregex "AD(C|D)S?ri", \
    “ANDS?ri", "BICS?ri", "CLZ", "EORri", "MVNS?r", "ORRri", "RSBri", \
    "RSCri", "SBCri")>;

…

// Floating Point Support
def : InstRW<[CarpWriteFPALU_F5, CarpRead_F1, CarpRead_F1], \
    (instregex  "(VADD|VSUB)(D|S|H)")>;
def : InstRW<[CarpWriteFPALU_F5, CarpRead_F1], \
    (instregex "VCVT(DS|SD|BHD|BDH|THD|TDH)")>;
def : InstRW<[CarpWriteFPST_F4, CarpRead_F0, CarpRead_F2], \
    (instregex "VSTR(D|S|H)")>;
def : InstRW<[CarpWriteFPDIV_SP, CarpRead_F0, CarpRead_F0], \
    (instregex "VDIV(S|H)")>;
def : InstRW<[CarpWriteFPDIV_DP, CarpRead_F0, CarpRead_F0], \
    (instregex "VDIVD")>;
```

Sched Model - Flow



ARM

Progress

LLVM Scheduler – What's present ?

- Easy to describe machine model
  - Per operand latencies
  - Resource utilization/cycles
- Compact with instructions inheriting from Sched
  - Consistency across sub-targets
- Ability to define variable number of writes
  - Varying Micro-ops, resource-cycles, latencies

LLVM Scheduler – What's missing?

- Instructions with slot constraints
  - Cannot issue in second slot – specification and pickNode changes
  - Cannot issue with any other – micro-ops
  - Cannot issue with specific another – reliance on resource constraint (not adequate)
- Inter-lock constraint modelling
  - Cannot slow down previous instruction
- First-half, second-half and in-stage forwarding
  - Further divide pipeline stages
- Variadic instructions
  - SchedPredicate, SchedVariant – an alternate compact representation necessary

Conclusion

- Significant performance improvement with accurate model
- Most pipeline features easy to capture
- Dynamic decisions by hardware hard to model
  - Not a limitation, but that's how it is
- Some aspects harder to describe in current tablegen descriptions
  - Continuous improvement, discussion points

ARM