Customized InOrder CPU Modeling

Korey Sewell
University of Michigan, Ann Arbor
(Now at Qualcomm)

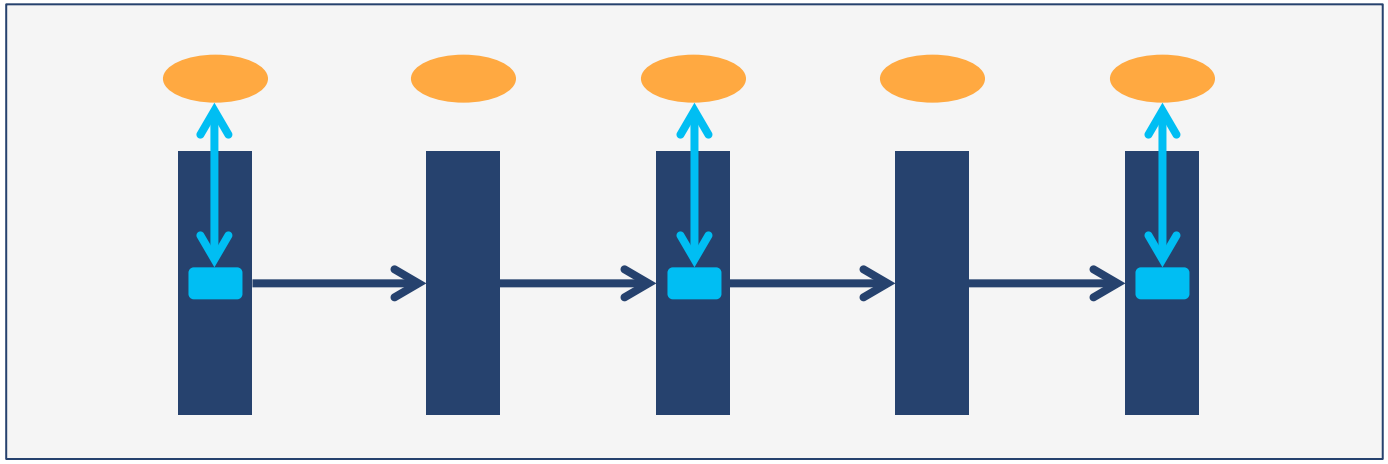
December 2nd, 2012

Customized InOrder CPU Modeling

- Agenda
  - InOrder CPU Model Basics
    - Pipeline Stages
    - Resources
    - Instruction Scheduling
  - Custom Resource Example: “Simple Value Predictor”
    - Deriving a Custom Resource
    - Making a Resource Recognizable
    - Adding a Resource to an Instruction Schedule
  - Extras
    - Complex Resource Handling, Stats, and Debugging

InOrder CPU Model Basics

- The InOrder model was designed to be a generic, flexible framework for CPU simulation:
  - The user defines the number of ***Pipeline Stages***
  - The user defines the ***Resources*** to be used in the pipeline (e.g. Branch Predictors, ALUs, etc.)
  - Each ***Instruction*** tells the current Pipeline Stage what resources it needs to access before it can be sent to the next stage.
    - The full list of actions that an instruction needs from each resource is called an ***Instruction Schedule***

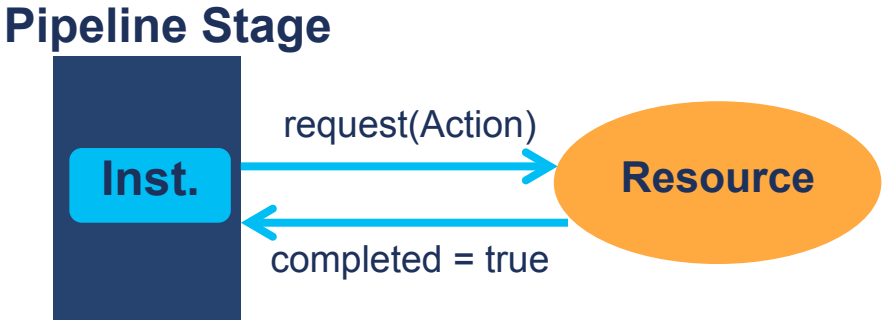


InOrder CPU Model

InOrder CPU Basics: Pipeline Stages

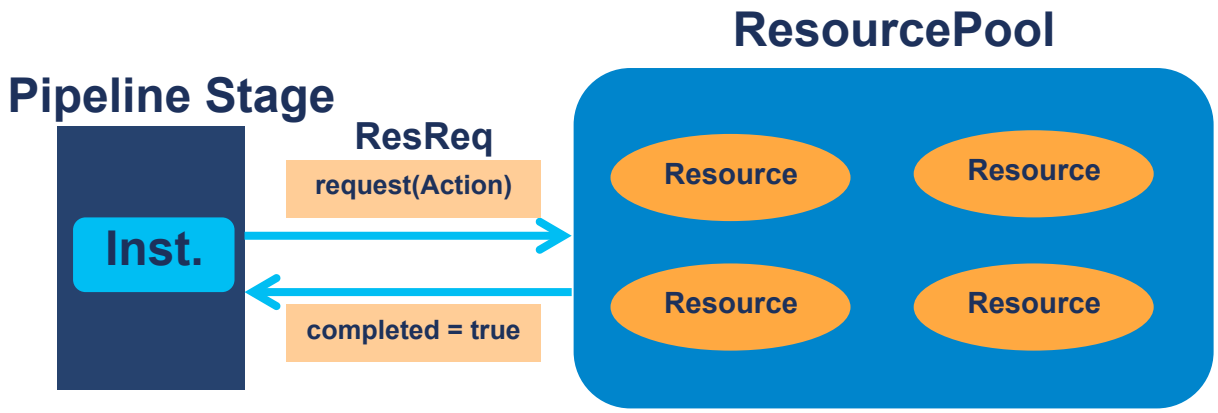
- The *PipelineStage* class provides a generic structure for instructions to communicate with resources
  - Implementation(s): `pipeline_stage.cc, first_stage.cc`
  - Special Case: *FirstStage* is a derived class whose sole purpose is to create instructions for the pipeline
  - Pseudo code:

```
for each Instruction in the pipeline stage
        for each Resource that an instruction needs to access
                bool completed  = request(Instruction, Resource, Action);
                if (!completed)
                        Stall the Pipeline;
                        Stop Processing Instructions for this Cycle;
        Send Instruction to the Next Stage's Buffer;
```



InOrder CPU Basics: Resources (1)

- A *Resource* represents a pipeline component that takes action for an instruction
  - Instructions request an action from a resource
  - Resources confirm/deny the completion of that action
- Implementations: `resource.cc`, `resource_pool.cc`, `resources/*.cc`
  - A *ResourceRequest* (ResReq) transfers data between instruction and resource
  - Resources are instantiated and accessed through the *ResourcePool* interface



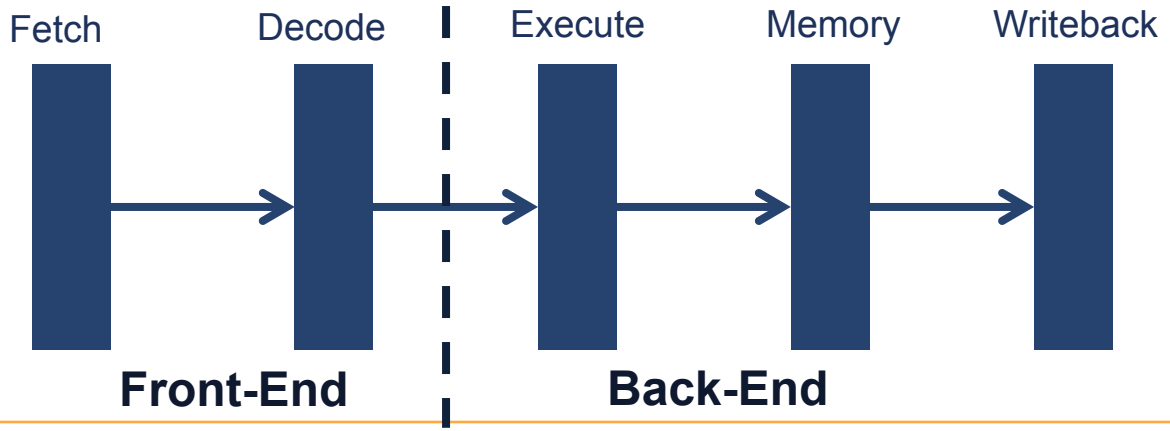
InOrder CPU Basics: Resources (2)

- Each resource defines the following parameters:
  - width – how many instructions that can be simultaneously processed
  - latency – cycles before the requested action is finished in the resource
- Custom Resources derive from the *`Resource`* base class and must also define the following:
  - `enum Command { Action1, Action2, ... };`
  - `void execute(...);`
- CacheUnit::execute(...) pseudocode:

```
void CacheUnit::execute(...) {
        ...
        switch (Cache-Unit Command) {
                case InitiateRead:
                        ...
                case InitiateWrite:
                        ...
        }
...
}
```

InOrder CPU Basics: Instruction Schedules

- The *ResourceSked* class contains the list of action and resource pairings for each instruction
- Implementation: *resource_sked.cc*
  - The InOrderCPU creates a front-end and back-end schedule for each instruction
    - Front-End Sked: Created in *FirstStage*, all instructions follow this schedule (e.g., fetch and decode)
    - Back-End Sked: An instruction schedule based on the instruction type (e.g., int, fp, ld/st, etc.)
    - The variable *"BackEndStartStage"* is defined in pipeline_traits.hh



Custom Resource Example

- Goal: Predict the value of a source register when it is not readily available from the register file (e.g. during a cache miss)

- Plan: Create a "Simple Value Predictor" Resource
  - Keep a record of PC and source register values when instructions graduate (commit)
  - Use this record when an instruction's source register value isn't ready…

- Caveats:
  - How do we take care of misspeculation?
  - This is unrealistic in terms of implementation and we could do a lot better at value prediction … but it's just an example ☺

Defining a Custom Resource (1)

- Step 1: Derive *SimpleValPredictor* from *Resource*
  - Add 'ValStore' and 'ValLookup' Actions
  - Define "SimpleValPredictor::execute()" to use these actions
    - Hint: Use 'resources/agen_unit.hh,cc' as a code template

src/cpu/inorder/resources/simple_val_predictor.hh

```
#ifndef SIMPLE_VAL_PRED_HH
#define SIMPLE_VAL_PRED_HH
...
class SimpleValPredictor : public Resource {
     ...
     enum Command {ValStore, ValLookup };
     void execute(int slot_num);
     ...
     struct SrcRegs {
          int r1, r2;
     };
     std::map<Addr, SrcRegs> valMap;
};
#endif
```

src/cpu/inorder/resources/simple_val_predictor.cc

```
...
void
SimpleValPredictor::execute(int slot_num) {
     ResourceRequest *req = reqs[slot_num];
     switch (req->cmd) {
          case ValStore:
               //Code to Save SrcRegs
          case ValLookup:
               //Code to Lookup SrcRegs
     };
     // Mark as Completed
     req->done();
}
...
```

Defining a Custom Resource (2)

- Step 2: Make the InOrder CPU recognize *SimpleValPredictor*
  - Add header file to Resource list
  - Add identifier to *PipelineTraits* namespace
  - Instantiate *SimpleValPredictor* inside the *ResourcePool* constructor

src/cpu/inorder/resource_list.hh

```
...
#include src/cpu/inorder/resources/simple_val_predictor.hh
...
```

src/cpu/inorder/pipeline_traits.hh

```
...
enum ResourceId {FetchSeq, ICache, ... , ValPredictor};
...
```

src/cpu/inorder/resource_pool.cc

```
ResourcePool::ResourcePool(InOrderCPU *_cpu, ThePipeline::Params *params) {
        ...
        resources.push_back(new SimpleValPredictor("SimpleValPredictor",
                                                   ValPredictor, stage_width,
                                                   0, params));
        ...
}
```

Defining a Custom Resource (3)

- Step 3: Add *SimpleValPredictor* to Instruction Schedules
  - Add 'ValLookup' action to Execute Stage
    - Note: *UseDefUnit*, the resource responsible for managing the Register File, would also need to be edited to use the predicted value.
  - Add 'ValStore' action to Writeback Stage

src/cpu/inorder/cpu.cc

```
InOrderCPU::createBackEndSked(DynInstPtr inst) {
          ...
          StageScheduler X(res_sked, stage_num++);
          StageScheduler M(res_sked, stage_num++);
          StageScheduler W(res_sked, stage_num++);
          ...
          // EXECUTE
          X.needs(ValPredictor, SimpleValPredictor::ValLookup);
          ...
          // WRITEBACK
          W.needs(ValPredictor, SimpleValPredictor::ValStore);
          ...
}
```

Extras: Complex Resource Handling

- Examples of more complex resource handling can be found in the “src/cpu/inorder/resources” directory
  - What if I need to pass extra information in my resource request?
    - First, try to save that information in the `DynInst` which is visible to any `Resource`.
    - Derive a new Resource Request type
      - See `UseDefRequest` in “resources/use_def_unit.cc” and it’s usage in `InOrderCPU::createBackEndSked()`
  - What if I want to define a multi-cycle resource?
    - Define ‘InitiateAction’ and ‘CompleteAction’ commands in your resource so that you can split the execution amongst two requests.
      - Consider using these action commands in different stages to hide latency.
    - See how this can be done by referencing the *`CacheUnit`* and/or the `MDU` code.

Extras: Custom Stats and Debugging

- How do I add stats to my Resource?
  - Declare a “Statistics::” variable in your class and register it with the `regStats()` function.
  - See `resources/agen_unit.hh,cc` for an example

- Where do I add debugging information for my Resource?
  - Add a “DebugFlag” in `src/cpu/inorder/SConscript`
  - Make sure to include the header for your DebugFlag
    - This will get automatically generated via the SConscript definition
  - Use the DebugFlag in the standard gem5 DPRINTF messages:

```
#include “debug/SimpleValPredictor.hh”
...
DPRINTF(SimpleValPredictor, “<Insert Message Here>”);
...
```

Fin.