Out-of-Order SMIPS

6.375 Microarchitecture Design
6 April 2011

Team: David S. Greenberg and Bhaskar Mookerji

Abstract

For our 6.375 final project, we are implementing a out-of-order processor using Tomasulo's algorithm and a reorder buffer. The following extends our high-level design to microarchitecture interfaces and discusses, in detail, our pipeline functionality and current implementation status. Our current implementation is available online at `https://github.com/mookerji/finalproject_6375`.

1 Tomasulo's Algorithm at a Glance

Before getting into the details of Tomasulo's algorithm, let's try to understand it at a high level. An instruction set architecture (ISA) has a finite number of places in which it can store variables. These places are called registers. Tomasulo's algorithm dynamically computes the dependencies between instructions so that instructions that don't have data dependencies can run in parallel, while only the truly data-dependent instructions are run in sequence. This increases the throughput of the system by exploiting instruction level parallelism (ILP). Tomasulo's algorithm can be seen as a dynamic hardware implementation of Static Single Assignment, a compiler technique that does the same dependency calculations in software.

The core idea of Tomasulo's algorithm is that each ISA register has a corresponding physical register. Since instructions are dispatched in order but executed out of order, we can associate the source operands of the instruction with the physical registers corresponding to the ISA registers used in the instruction. The destination register of the instruction, however, is written to a new physical register, and the destination ISA register is updated to be linked to the new physical register. That way, every register is assigned once and only true dependencies affect execution.

Of course, hardware can't be allocated, and so Tomasulo's algorithm allocates the physical registers from a pool and releases registers back into the pool when there are no more instructions using them as a source operand. The details of this are explained later.

2 Problem

Although an elastic pipelined microprocessor is an improvement over an unpipelined processor, its performance suffers due to pipeline stalls. There are three classes of stalls, two of which are eliminated by Tomasulo's algorithm.

The first class of stall is a Write-After-Read stall. This is where we see an instruction A which reads from register r_x followed by an instruction B which writes to the same register r_x.

Cycle	1	2	3	4	5	6	7	8
Multiply	fetch	exec 1	exec 2	exec 3	exec 4			
Move		fetch	stall	stall	stall	exec		
Add			fetch	stall	stall	stall	exec 1	exec 2

Table 1: Snippet 1 executed on a simply pipelined microprocessor

Normally, B's execution is blocked by A since they both need to access the same register. If A has several cycles of latency, then B will be stalled until A completes execution. Tomasulo's algorithm eliminates this false dependency by writing the result of B to a separate physical register, allowing B to execute in parallel with A.

The second class of stall is a Write-After-Write stall. This is where we see an instruction C which writes to a register r_y followed by an instruction D which writes to the same register r_y. Normally, since both instructions need to write to the same ISA register, D would be stalled until C completed; however, Tomasulo's algorithm allows them to execute in parallel since the instructions would each write to separate physical registers.

The third class of stall is a Read-After-Write stall, or a true data dependency. This is where instruction E writes to register r_z, and then instruction F reads from register r_z. Tomasulo's algorithm does nothing in this case, because since F needs the result of E, it must be stalled until E completes.

Let's look at an example snippet of code to understand an example situation in which Tomasulo's algorithm would help (Snippet 1). In this simplified assembly language, the format of an instruction is *op*, src_1, [src_2], *dst*; *src*2 is optional.

```
mul r1, r2, r3
mov r3, r4
add r1, r2, r3
```

Snippet 1: Instruction sequence which would benefit from Tomasulo's algorithm

Let's assume that multiplies take 4 clock cycles, moves take 1 clock cycle, and adds take 2 clock cycles. Also, let's assume that fetching an instruction takes 1 clock cycle and happens in parallel with execution. Then executing these instructions in sequence takes a total of 8 clock cycles–one cycle for the initial fetch, followed by each instruction executing in sequence (Table 1). Since fetches are pipelined, we only see the fetch latency on the initial instruction. Now, if we are using Tomasulo's algorithm, we'll instead see a total runtime of 6 clock cycles. To understand why, we can look at this table 2.

3 High-Level Design

The system design and instruction pipeline for our implementation is described in Figure 1(a) and 1(b), respectively. The design extends our existing SMIPSv2 implementation into three

Cycle	1	2	3	4	5	6
Multiply	fetch	exec 1	exec 2	exec 3	exec 4	
Move		fetch	stall	stall	stall	exec
Add			fetch	exec 1	exec 2	

Table 2: Snippet 1 executed on a microprocessor with Tomasulo's algorithm

components: in-order instruction fetch and decoding, out-of-order execution units, and a commit unit. The instruction fetch and decode stage enqueues and dispatches instructions to reservation stations matching the appropriate execution unit, reordering as necessary to remove the pipelining hazards described in Section 2. When the reservation station contains the appropriate operands for a given instruction, execution proceeds, and results are broadcast on a common data bus to waiting reservation stations and the branch prediction unit. The commit unit is implemented as a reorder buffer, and performs in-order commits to memory and the register file. Taken together with dispatch and register renaming, the reorder buffer and the reservation stations implement Tomasulo's algorithm. The details of this implementation are described in [KMP99] and [HP07].

4 Goals and Testing Strategy

4.1 Functional Correctness

We will use the existing SMIPS implementation and toolchain to provide inputs, gather outputs, log rule executions as traces, and benchmark our implementation. Verifying the correctness of the Tomasulo's algorithm implementation will first require modular refinement of fundamental components, such as the reorder buffer and reservation stations, before they are incorporated into our existing SMIPS implementation. To formally verify functional correctness, we can examine traces for two different areas: data consistency invariants and instruction completion/termination/stalling conditions[1].

To further check for bugs, we will unit test with short programs that emphasize particular issues: sequences of dependent instructions, sequences of independent arithmetic instructions, small loops, atypical branch sequences, and memory operations. Verifying the traces for these smaller test programs will simplify execution debugging for the existing benchmarks.

4.2 FPGA Synthesis

We've removed all CAM from the system, and so we believe that we should be able to synthesize modulo routing constraints.

[1]This criteria for functional correctness in Tomasulo's algorithm are described in greater detail in Chapter 6 of *Design and Evaluation of a RISC Processor with a Tomasulo Scheduler* at `http://www.kroening.com/diplom/diplom/`

4.3 Performance

Raw performance is not the goal of this project. With this architecture, we might not reach the IPCs of the three-stage pipeline architecture in Lab 4. To demonstrate the improvement of out-of-order execution in our processor, we will artificially introduce latency into our execution units using pipelined integer arithmetic routines. Theoretical performance and correctness are the primary concerns. Our design should enable future developers to use a modularized Tomasulo's algorithm to hide the latencies of their additional instructions. Unfortunately, there is way to show a strict benefit to the elastic pipelined SMIPS processor, since all ALU ops and multiplies are single cycle on an FPGA anyway, and any worthwhile multicycle operation is a project in and of itself to implement.

5 Microarchitecture Overview

The following describes, in detail, newly implemented modules and extensions that facilitate out-of-order execution using the Lab 5 and 6 SMIPS processor. Section 5.1 describes new types introduced in `TomasuloTypes.bsv`, Section 5.2 provides pseudocode describing rule execution for a five-stage elastic pipeline in `Processor.bsv` and `ALU.bsv`, and Section 5.3 summarizes the architectural modules that contain these types during the pipeline execution. Due to the ad-hoc organization of this document, we recommend you read Sections 5.1, 5.2, and 5.3 in-parallel. A microarchitecture diagram indicating the types, rules, modules, and interfaces in shown in Figure 2.

5.1 New System Types

1. **Reorder Buffer Tag and Reorder Buffer Entry**. At instruction issue, a reorder buffer tag (`ROBTag`) is generated by a token request at the reorder buffer. Keeping track of this value eliminates the need for associative lookup by providing an index directly into the ROB. The ROB itself contains the entries (`ROBEntry`) with the intended register destination, a `Maybe#{data}` for a result, and mispredict and epoch fields for handling branch prediction. We have augmented the original `Rindx` type by defining `WBReg` to also point to a `MFC0` or `MTC0` special purpose register.

```
typedef Bit#(4) ROBTag;

typedef union tagged
  Rindx ArchReg;
  CP0indx SpecReg;
 WBReg deriving (Bits, Eq);

typedef struct
        Maybe#(Data) data;
```

```
        Addr pc;
        Maybe#(Addr) mispredict;
        WBReg dest;
        Epoch epoch;
 ROBEntry deriving (Bits, Eq);
```

2. **Rename Entry**. The rename entries are contained in the rename register file, which extend the register file by indicating if the register file contents are valid, and if not, where the data might be found through the reorder buffer.

```
typedef union tagged
    void Valid;
    ROBTag Tag;
 RenameEntry deriving (Bits, Eq);
```

3. **Reservation Station Entry**. Entries for the reservation station (`RSEntry`) store the operands, the ROB tag of the instruction in the reservation station, and an instruction tag. The instruction tag is used for dispatching during issue and execution, and may also be extended by a immediate field if two other operands already exist. An `Operand` is either a data immediate or a tag for the ROB instruction that produces the desired value. An `Op_type` is used later for dispatching classes of instructions.

```
typedef union tagged
    ROBTag Tag;
    Data Imm;
 Operand deriving (Bits, Eq);

typedef struct
        InstrExt op;
        ROBTag tag;
        Operand op1;
        Operand op2;
        Addr pc;
        Epoch epoch;
 RSEntry deriving (Bits, Eq);

typedef enum ALU_OP, MEM_OP, JB_OP, MTC0_OP Op_type deriving(Eq);

typedef union tagged
    struct  LW;
    ...
```

```
    struct  Simm offset;   BEQ;
    ...
 InstrExt deriving (Bits, Eq);
```

4. **Common Data Bus Packet**. The CDB's bypasses a packet to reservation station entries, containing a data immediate, its appropriate operand tag, and an epoch for branch prediction.

```
typedef struct
    Maybe#(Data) data;
    ROBTag tag;
    Epoch epoch;
 CDBPacket deriving (Bits, Eq);
```

5. **ALU Request and Response**. The `ALUReq` and `ALUResp` are passed to the ALU module. The `ALUReq` uses a `InstrExt` field to dispatch operations and bypasses the ROB tag through to the completion and writeback pipeline stages. Arithmetic operations are performed on `op1` and `op2` to yield `ans`. Branch and jump operation targets are calculated and passed through the ALU, so this type has been augmented to include this information.

```
typedef struct
    InstrExt  op;
    Data      op1;
    Data      op2;
    ROBTag    tag;
    Addr      pc;
    Epoch     epoch;
 ALUReq deriving (Eq, Bits);

typedef struct
    InstrExt  op;
    Data   data;
    ROBTag tag;
    Addr   pc;
    Addr   next_pc;
    Epoch  epoch;
 ALUResp deriving(Eq,Bits);
```

5.2 Pipeline Stage Behavior

The rules in `Processor.bsv` implement an elastic pipeline intermediated by the container modules described in Section 5.3. Branch prediction handling and stalling logic are included, but omitted from the pseudocode. Some pseudocode notation are lifted from [KMP99].

1. **Instruction Fetch**. During instruction fetch, the instruction is fetched and enqueued, and the `pcQ` is enqueued with the current epoch and program. This is implemented by the `fetch` rule.

2. **Instruction Decode and Issue**. The instruction is interpreted and dispatched to the appropriate reservation station. During the decode stage, the instruction is piecewise parsed into an instruction tag, a possible instruction tag extensions, and its operands (either register sources or immediates). The decode stage's primary job is to resolve the operand sources and values of a given instruction I_i.

```
if (∃ free RS for I_i and ROB.empty):
    RS.op =  I_i
    RS.tag = ROB.tail
    RS.epoch = pcQ.epoch
    RS.pc = pcQ.pc

    ∀ operands x of I_i:
        if (R_x.A is Valid):
            RS.op_x = Valid R_x.A.data
        else if (CDB.tag == R_x.A.tag):
            RS.op_x = Valid CDB.data
        else if (ROB[R_x.A.tag] is Valid):
            RS.op_x = Valid ROB[R_x.A.tag].data
        else
            RS.op_x = Invalid with R_x.A.tag

    if (I_i has desination register y.A)
        R_x.A.tag = ROB.tail
        ROB[ROB.tail].dest = y.A
    else
        ROB[ROB.tail].dest = 0
```

   If the reservation station (RS) and reorder buffer (ROB) are available, a reservation station entry is initiated, grabbing a ROB token. Here, the register address of operand x is denoted by $x.A$. The decode stage instantiates the RS entry by checking in three places: the register file, the common data bus, or the reorder buffer. If Invalid, the instruction tag is stored in the reservation station. Simultaneously, the reorder buffer

is updated with the issued instruction. This is implemented by the `decode_issue` rule. Note that memory operations are currently handled identically to lab 5 and 6: we wait until the reorder buffer is empty, meaning that no instructions are in-flight, before we issue load and store operations.

3. **Dispatch**. During dispatch, the operands and instruction tag are passed for execution.

```
∀ operands x
    if (RS.op_x tagged Invalid with .tag and tag == CDB.tag)
        RS.op_x = Valid CDB.data
```

The instructions currently reside in the reservation station, which is constantly snooping the CDB to update it's operands. Once all operands are valid and the functional unit is ready to accept a new instruction, the functional unit is initialized.

```
if (∃ free RS with Valid RS.op_x ∀
    operands x and !FU.stall):

    FU.op = RS.op
    FU.tag = RS.tag
    FU.op_x = RS.op_x
```

The snooping is an implicit guard implemented by the reservation station entry module. The rest is implemented by the `dispatch_alu` and `dispatch_mem` rules

4. **Execute**. The arithmetic operation or memory operations are performed. This is described in greater detail in the ALU module details, although the dispatch and complete rules interface with `aluReqQ` and `aluRespQ` FIFOS, which have direct connections to the ALU module.

5. **Completion**. The result of the execution is bypassed and updated into the ROB. The reservation station requests the CDB, the result and the tag are placed on the CDB, and the matching ROB entry is tagged valid with the result.

```
if (FU has result and got CDB-ack):
    CDB.data = FU.result
    CDB.tag = FU.tag
    ROB[CDB.tag] = Valid CDB.data
```

This is implemented by the `alu_compl` and `cdb_rob_bypass` rules. Instructions are completed only if they match the current epoch, otherwise they are purged by the `purge` rule (and the rename file is updated accordingly). If the instruction is a jump or branch instruction and the prediction is wrong, it is updated in the branch predictor.

6. **Graduate**. The ROB writes back the execution result to the register file.

```
if (!ROB.empty and ROB[ROB.head] is Valid):
    x = ROB[ROB.head].dest
    R_x.data = ROB[ROB.head].data
    if (ROB.head == R_x.tag):
        R_x = Valid
```

   This is implemented by the `graduate` rule.

5.3 Architectural Interfaces

The following describes the support architectural modules implemented in our design.

1. **Reservation Stations**. The reservation stations enqueue the instructions, their operands, and any appropriate tags, as an `RSEntry`, until a functional unit is available. As soon as all the operands are available, they are dispatched to the actual functional unit. The reservation station module keeps track of when its operands are ready for evaluation. The `put` method enqueues an `RSEntry`.

```
interface ReservationStation;
  method ActionValue#(RSEntry) getReadyEntry();
  method Action put(RSEntry entry);
endinterface
```

2. **Common Data Bus**. Execution units place their results, packed as `CDBPacket`s, on the common data bus (CDB) module. Currently, this action can only be done once in a single clock cycle. Units reading from the CDB are called listeners. The reservation station snoop the CDB for missing operand immediates. All listeners must check the bus every cycle and must acknowledge before accepting a new value. The CDB also has a state method `hasData` for enforcing pipeline guards during CDB snooping.

```
interface CommonDataBus#(type t);
  method Action put(t entry);
  method ActionValue#(t) get0();
  method ActionValue#(t) get1();
  method ActionValue#(t) get2();
  method ActionValue#(t) get3();
  method Bool hasData();
  method Action dumpState();
endinterface
```

3. **ALU**. The ALU module accepts an `ALUReq` type request and responds with a `ALUResp` type response using a `proc_server Server#(ALUReq, ALUResp)` subinterface. This module is only used when two operands are available and dispatched from the ALU's reservation station.

```
interface ALU;
    interface Server#(ALUReq, ALUResp) proc_server;
endinterface
```

4. **Reorder Buffer**. The ROB module is just great: it ensures that completed instructions are retired in program-order. It is implemented as a circular FIFO queue with head and tail pointers. Newly issued instructions are put into the ROB entry indexed by the tail pointer, whose value is also used to tag the result. By bypassing these tags through the pipeline stages, we use an ROB without the synthesis overhead of associative lookup. Updating of the head and tail pointers is handled internally, as well as the retiring of ROB entries upon register writeback, with the `getLast` and `complete` methods. The `isEmpty` and `isFull` methods indicate the state of the ROB. The `reserve`, `update`, `get` methods return a tag, update a tagged entry, and retrieve a tagged entry, respectively. The ROB module is also used in purging instructions of the wrong epoch and issuing a mispredict if a branch has been mispredicted.

```
interface ROB#(numeric type robsize);
  method ActionValue#(Bit#(TLog#(robsize))) reserve(Epoch epoch,
                                                    Addr pc,
                                                    WBReg dest);
  method Action updatePrediction(Bit#(TLog#(robsize)) tag,
                                 Addr mispredict);
  method Action updateData(Bit#(TLog#(robsize)) tag, Data data);
  method Maybe#(ROBEntry) get(Bit#(TLog#(robsize)) tag);
  method ROBEntry getLast();
  method Bit#(TLog#(robsize)) getLastTag();
  method Action complete();
  method Bool isEmpty();
  method Bool isFull();
endinterface
```

5. **Branch Predictor (BTB).** The branch predictor unit is implemented as branch target buffer. It stores the current epoch, returns a prediction given it's current state, confirms a prediction, and alters state based on misprediction.

```
interface BranchPredictor#(numeric type btbsize);
 method ActionValue#(Addr) predict();
 method Addr confirmPredict(Addr currentPc);
 method Epoch currentEpoch();
 method Action mispredict(Addr srcPc, Addr dstPc);
endinterface
```

6 Implementation Status and Testing Plans

We have implemented every component of the processor. We have begun to debug the system, and can verify that instructions are being issued and dispatched correctly into the relevant functional units (reservation stations, reorder buffer, common data bus, etc.). During execution, we can trace instructions through the pipeline and determine where substantial pipeline stalls occur. Currently, we execute around 40 instructions, including several branches and loops before running into issues in the qsort benchmark. We can pass around 15% of the asm testbench.

7 Planned Design Exploration

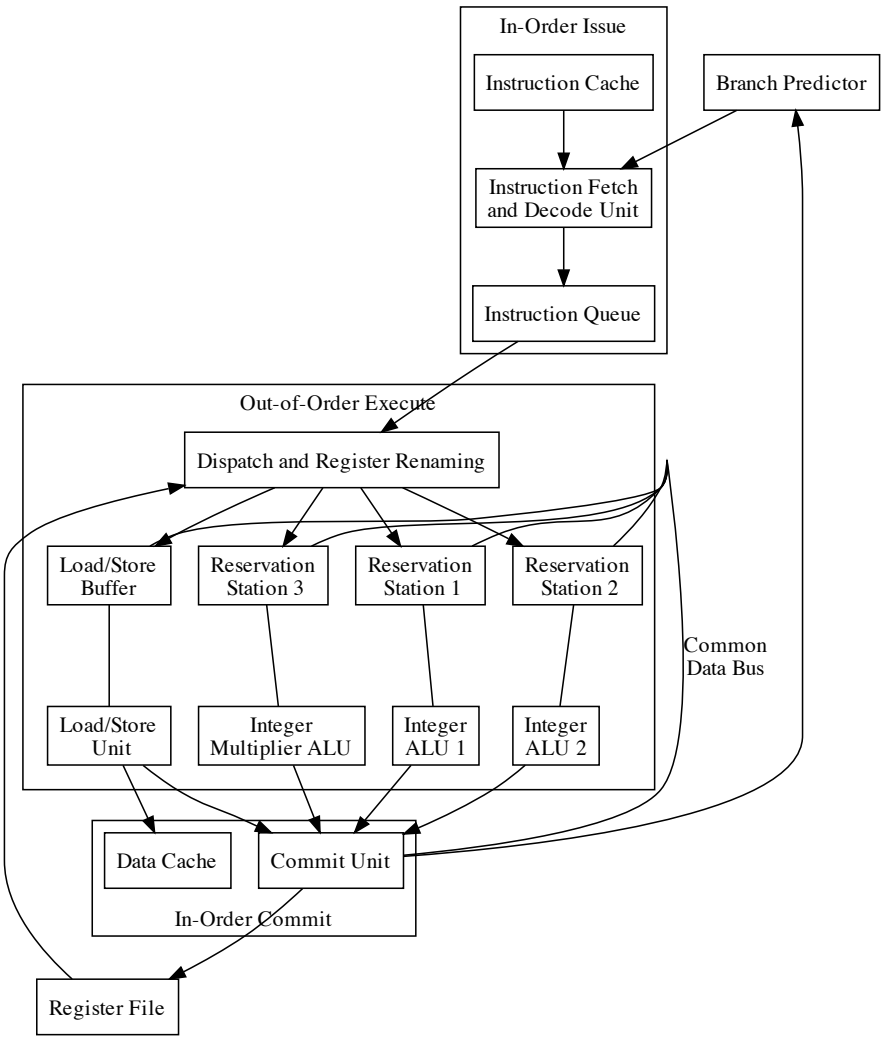
Our first goal is to obtain functional out-of-order execution with speculation, and then improve IPC performance by optimizing rule/pipeline concurrency and FIFO choices (as in Lab 6). We will explore architectural parametrization and design choices on the IPC and timing critical path:

1. Reorder buffer dimension
2. Branch prediction strategies (BTB and 2-Bit)
3. Register file bypassing and FIFO usage
4. Multiple ALU instruction issue
5. Artificial arithmetic and memory latency

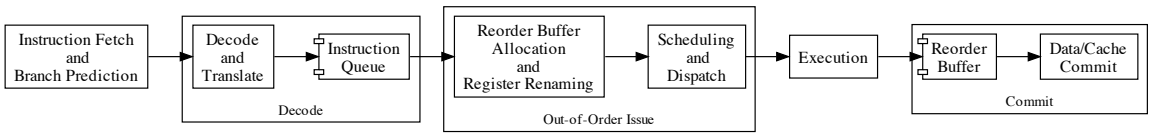
References

[HP07] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach (4. ed.)*. Morgan Kaufmann, 2007.

[KMP99] Daniel Krning, Silvia M. Mller, and Wolfgang Paul. A rigorous correctness proof of the Tomasulo scheduling algorithm with precise interrupts. In *Proc. of the SCI'99/ISAS'99 International Conference*, 1999.



(a) System architecture using out-of-order execution and integer arthmetic unit.



(b) Instruction pipeline. Pipeline stage is labeled in-box, unless superseded by an overbox. A component box indicates component (such as a FIFO) between stages. Not included in this diagram is the reservation stations and load/store buffer in the execution stage.

Figure 1: High level design elements.

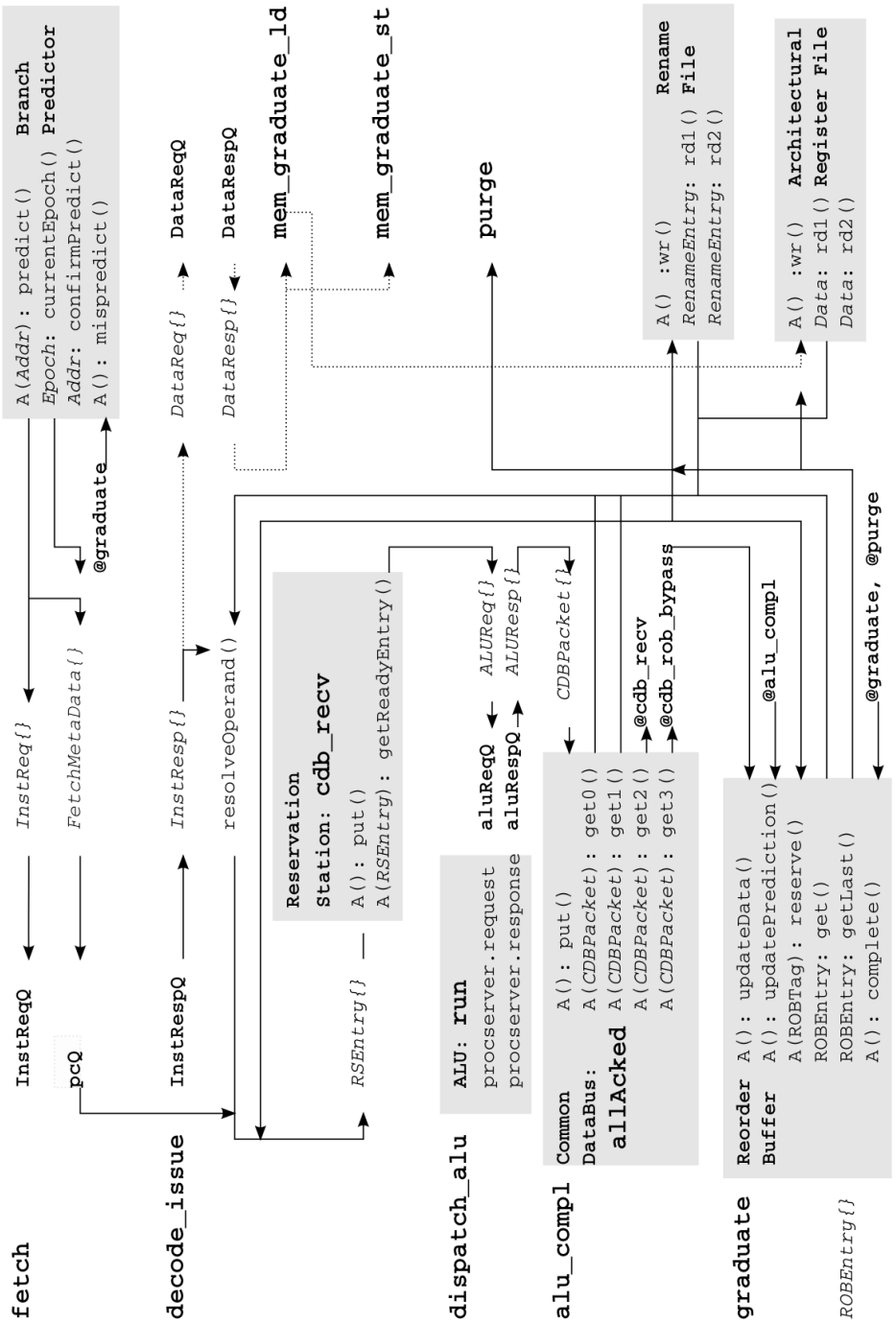


Figure 2: Microarchitecture details. This outlines the contents of Exec.bsv. The five main pipeline stages are denoted in large bold font on the left, and the area immediately to the right describes the rough describes the behavior within these rules. Types and methods are clearly indicated, and A( ) indicates an Action and ActionValue. Memory operations are executed in-order without the reorder buffer; the dataflow in this pipeline is indicated by a dotted line.