MASE User's Guide
(part of SimpleScalar version 4.0)

Eric Larson and Todd M. Austin

{larsone, austin} @eecs.umich.edu

Advanced Computer Architecture Laboratory

University of Michigan

December, 2001

Part 1: MASE Overview

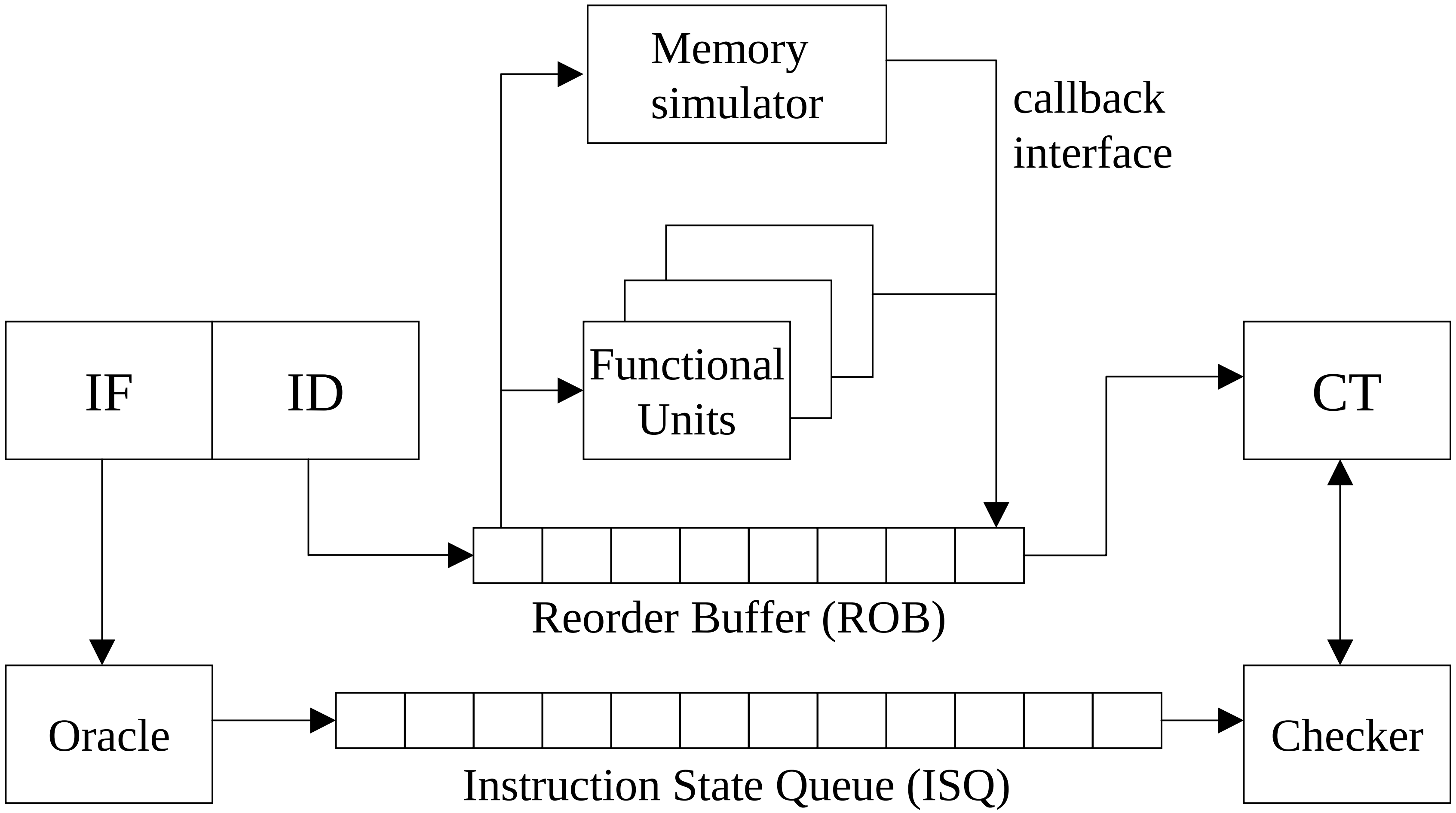
MASE overview

- MASE is a new performance simulation infrastructure for SimpleScalar. It serves as a replacement to sim-outorder.

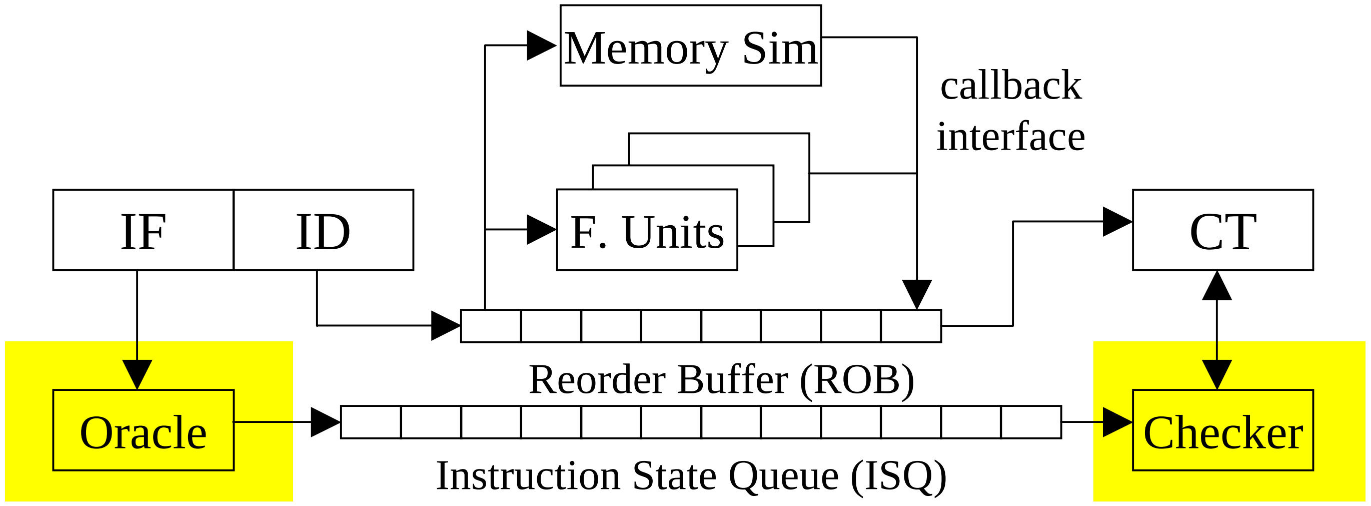
- Summary of new features in MASE:
  - Checker improves validation support.
  - Oracle allows for “perfect” studies.
  - Micro-functional performance model increases accuracy.
  - Speculative state management facilities simplify aggressive speculation.
  - Callback interface permits sophisticated memory system simulation.

- Installation directions are found in the file INSTALL. The MASE simulator is called sim-mase.

MASE software architecture

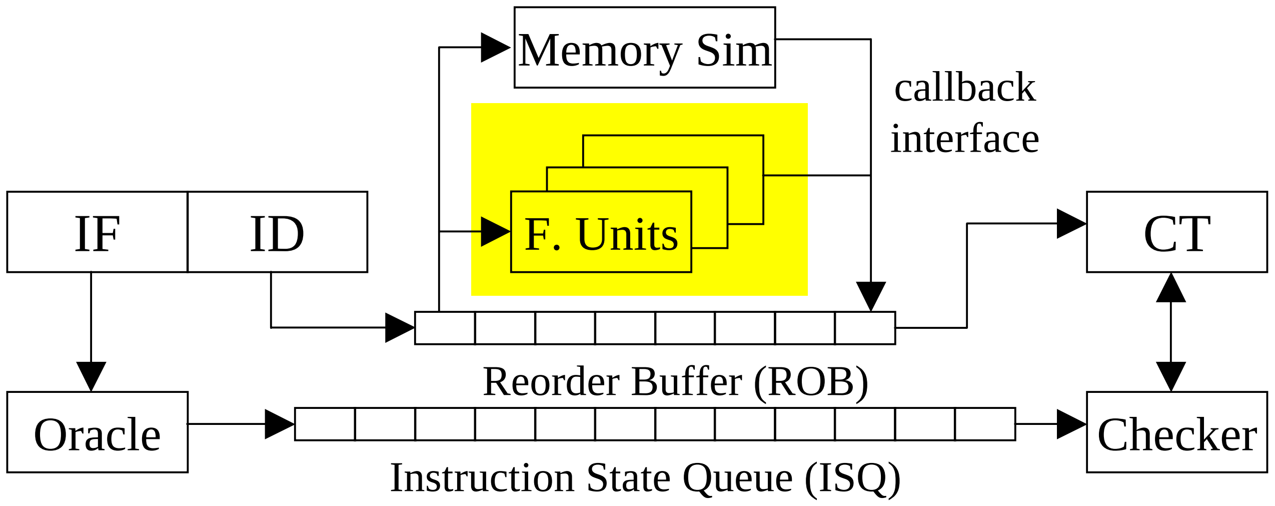


Checker and oracle



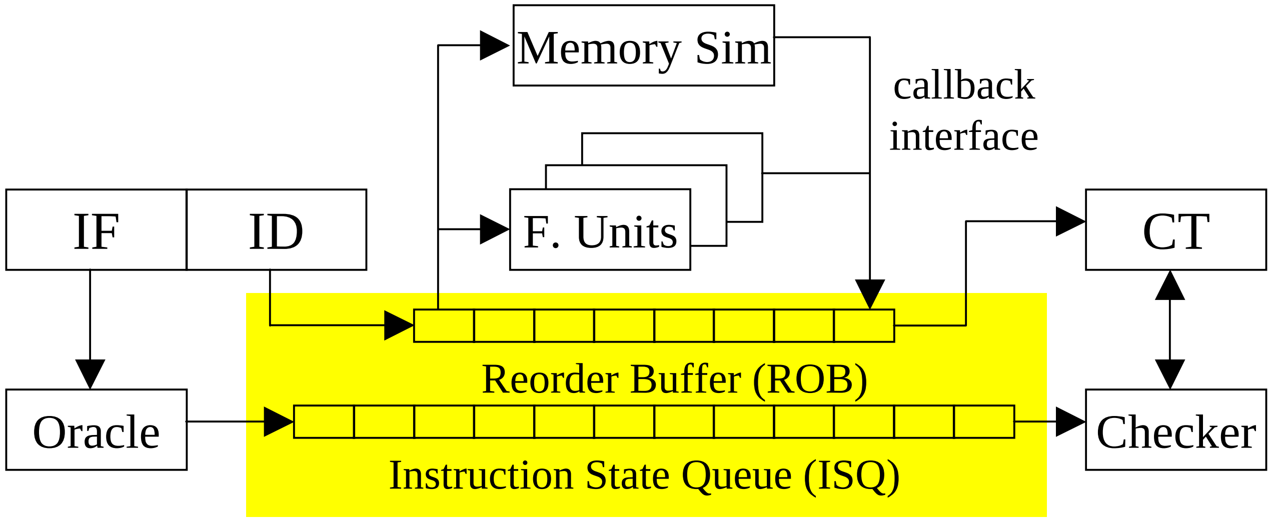
- Permit "perfect" studies and improved validation.
- Oracle executes in fetch and places values into ISQ.
- Checker uses ISQ values to validate core computation.
- Checker will fix any core bug, reducing burden of correctness in core.

Micro-functional performance model



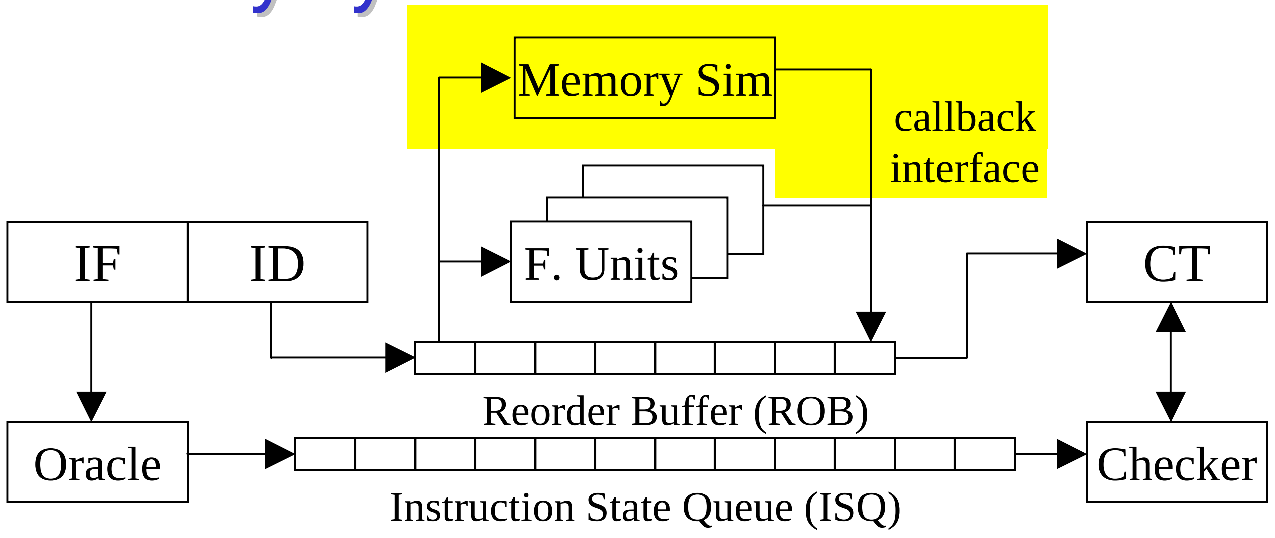
- Trace-driven techniques cannot accurately model timing-dependent computation.
  - For example, mispeculation and shared memory race conditions.
- Instructions are now executed in the core with proper timing.
- Further improves validation, intertwining timing and correctness.

Support for aggressive speculation



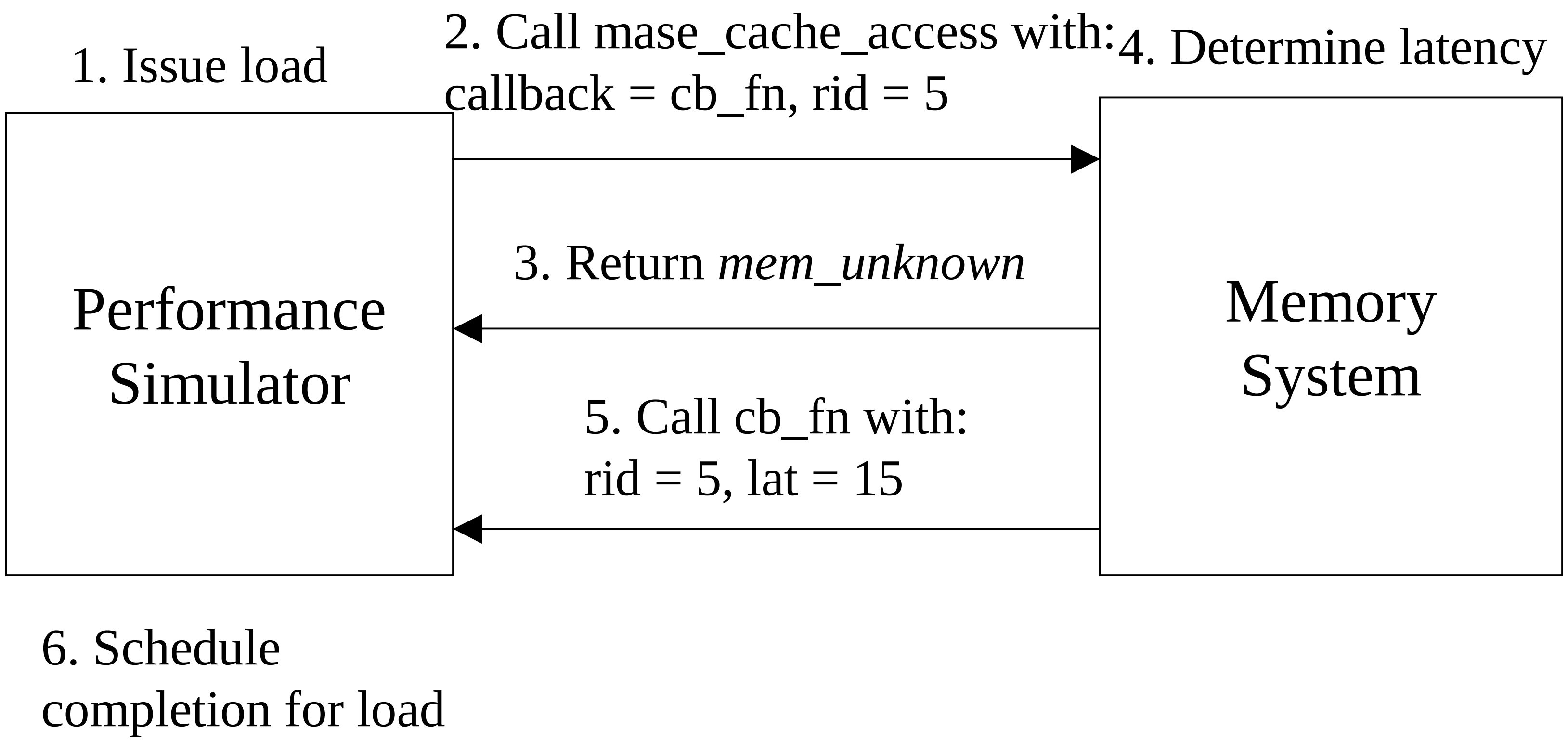
- SimpleScalar lacks arbitrary instruction restart. Only branches can restart.
- MASE allows any instruction to mispeculate and restart core.
- Several data structures (such as the ROB and ISQ) were modified to support arbitrary rollback.

Memory system with callback interface



- SimpleScalar’s memory system requires that instruction latency be known at issue.
  - Not representative of modern memory systems.
  - For example, DRAM accesses can be reordered to increase page hit rates.
- Instructions use callback interface to asynchronously declare their (remaining) latency.

Memory system with callback interface



Other improvements

- Algorithm for detecting when store data can be forwarded to loads has been improved (more aggressive).
- Register update unit (RUU) has been split into a reorder buffer (ROB) and reservation stations (RS).
- Added a scheduler queue.
  - Scheduler predicts the latency of each instruction.
  - Instructions are replayed if the prediction is too small.
- Added a front-end queue.
  - Improves misprediction delay accuracy.
  - Can simulate additional stages in the front-end pipeline.

Part 2: MASE Implementation

Summary of MASE files

- mase.c: main simulator loop, loading of program, fast forwarding
- mase-fe.c: fetch and dispatch, rename table management
- mase-exec.c: execution core, issue, lsq_refresh, event queue and ready queue management
- mase-commit.c: writeback, commit, some recovery functions
- mase-checker.c: oracle and checker
- mase-debug.c: functions for the DLite debugger
- mase-mem.c: memory interface functions
- mase-opts.c: option and statistic management
- mase-decode.h: decoding macros
- mase-macros-oracle.h: execution macros for oracle
- mase-macros-exec.h: execution macros for core
- mase-structs.h: common structure definitions

Fetch: mase_fetch() in mase-fe.c

1. Updates program counter (`fetch_PC` ← `fetch_pred_NPC`)
2. Fetch from I-cache, stalling on any miss. (only one cache line can be accessed per cycle).
3. Call the oracle to execute the instruction.
4. Access branch predictor to get next PC (`fetch_pred_NPC`).
5. Place fetched instruction into the instruction fetch queue.

- The steps above are repeated until a stall condition occurs. (fetch bandwidth met, cache miss, etc.)

Dispatch: mase_dispatch() in mase-fe.c

1. Checks to make sure there is an available reservation station, ROB entry, and LSQ entry (if it's a memory operation) to hold the instruction.
2. Fetch instructions from the IFQ.
3. Decode the instruction. Some decode information is pre-computed by the oracle and passed along in the IFQ.
4. Determine if there is a misfetch – the target address predicted by the BTB does not match the correctly decoded target address.
5. Initialize the ROB entry (and LSQ entry for memory operations) for the instruction.
6. Access the rename table to see where the instruction gets it input:

   architected register file: the value is read from directly from the register file

   creator has completed: the value is transferred from the creator's ROB entry.

   creator has not completed: instruction is added to the creator's output dependence chain.

- The steps above are repeated until a stall condition occurs. (decode bandwidth met, ROB is full, etc.)

Load Scheduling: lsq_refresh() in mase-exec.c

- Scans the LSQ starting at the head looking for loads that have their operands ready. Earlier stores can prevent execution of such loads.
- A store with an unknown address prevents all future loads from being scheduled (unless perfect disambiguation or blind load speculation is used). In perfect disambiguation, all stores have “known” addresses using oracle data.
- A store with a known address but unknown data blocks any future load that reads from the address. This is true for perfect disambiguation and blind load speculation.
- A store with a known address and known data, will transfer its store data to any future load that reads from that address.
- Loads that are not blocked and have their operands ready are placed into the ready queue.

Scheduling: ready_queue in mase-exec.c

- The ready queue serves as the instruction scheduler.
- Instructions are inserted into the ready queue when all of its input operands are ready. This occurs in either dispatch if the operands are ready immediately, at writeback when a dependent operation finishes, or in lsq_refresh which determines when loads are allowed to execute.
- Long-latency operations are inserted at the head of the queue to give them priority. Other instructions are inserted in program order with older instructions placed towards the head of queue.

Issue: mase_issue() in mase-exec.c

1. Check for delays due to latency mispredictions.
2. Get instruction from the head of the ready queue.
3. If the instruction requires a functional unit, a functional unit is reserved. If there is not an available functional unit, the instruction cannot issue and is placed back into the ready queue.
4. If the instruction is a load, the instruction will access the DTLB and D1 data cache.
5. Instruction is issued into scheduler queue. For instructions with deterministic latency, an event is scheduled at the time when the instruction completes (event_queue).
6. The reservation station used by instruction is reclaimed.
7. Issued instructions are executed in this routine but the output is not marked valid until the instruction completes (writeback).
8. Instructions that do not issue are placed by in the ready queue.

Execute: mase-macros-*.h

- Instructions are executed twice: Once by the oracle in checker_exec (mase-checker.c) and once by the core in execute_inst (mase-exec.c).
- The file machine.def indicates how to execute the instruction and is used by both the oracle and the core. The next slide tells more about the format of this file.
- machine.def uses macros such as that are defined in mase-macros-oracle.h and mase-macros-exec.c. Examples of the macros are:

`GPR(N)`	Returns the value of general purpose register N.
`SET_GPR(N,EXPR)`	Writes EXPR to general purpose register N.
`READ_WORD(ADDR, FAULT)`	Read word from memory at address ADDR.
`WRITE_WORD(ADDR, DATA, FAULT)`	Write DATA to memory at address ADDR.

- The macros for the oracle (mase-macros-oracle.h) are straightforward can directly read and write the pre-update register file and memory.

Execute: mase-macros-exec.h, mase-exec.c

- The macros for executing in the core (mase-macros-exec.h) access the values stored in the ROB_entry.
- For register read operations: The appropriate value from the idep_value array is returned. (see read_idep_list)
- For register write operations: The appropriate element from the odep_value array is written. (see set_odep_list)
- For load operations: If data was transferred via store forwarding, the value is in mem_value and is merely returned. Otherwise, architected memory is accessed to return the proper value. (see read_load_data)
- For write operations: The store data is placed into the mem_value field of the ROB_entry. (see read_store_data)
- NOTE: This implementation causes the following restrictions on instruction definitions in machine.def:
  - Registers cannot be read a register once a value has been written.
  - Output registers cannot be read unless they are specifically indicated as an input.
  - There can be at most one memory (read or write) operation per instruction.

Execute: machine.def

- Used to define the instructions including dependencies and semantics.
- Can be used to generate decoders, dependency analyzers, functional components, disassemblers, appendices, etc.
- Instruction definition example:

semantics

```
#define ADDQ_IMPL {                       \
  SET_GPR(RC, GPR(RA) + GPR(RB));   \
}
DEFINST(ADDQ,              0x00,
        “addq”,             “a,b,c”,
        IntALU,             F_ICOMP,
        DGPR(RC),DNA,       DGPR(RA), DGPR(RB), DNA)
```

opcode

assembly template

assembly instruction

inst flags

FUs required

output deps

input deps

Writeback: mase_writeback() in mase-commit.c

- Looks at each instruction that completes that cycle.
- The outputs created by the instruction are marked valid.
- If the instruction is a mispredicted branch, a recovery occurs (more information on recovery later).
- The output dependence chain of the instruction is traversed
  - Mark the input operand for each dependent instruction as ready.
  - If all dependencies of a dependent instruction are now satisfied, the instruction is ready to execute and placed into the ready queue. Note: This does not occur for loads as memory dependences need to be checked which occurs later in lsq_refresh.

Commit: mase_commit() in mase-commit.c

1. Checks to see if the oldest instruction in the ROB has completed. If the oldest instruction is a memory operation, the corresponding LSQ entry must also be complete.
2. Store instructions access the TLB and data cache.
3. The instruction is checked by the checker and the results are committed to architected state. If the results do not match the checker, the oracle data is committed to architected state and a recovery occurs.
4. The rename table entry is removed.
5. The ROB and LSQ entries are all cleared and reclaimed.

Main Simulator Loop: sim-main() in mase.c

```
for (;;) {
  mase_commit();
  mase_writeback();
  lsq_refresh();
  mase_issue();
  mase_dispatch();
  mase_fetch();
}
```

- Walks pipeline from Commit to Fetch.
  - Backward pipeline traversal eliminates relaxation problems providing correct inter-stage latch synchronization
- Loop is exited via a `longjmp()` to `main()` when simulated program executes an `exit()` system call or the function simply returns when the maximum number of instructions has been met.

Part 3: MASE Options

Global Simulator Options

- These options are supported on all simulators:

`-h`	print simulator help message
`-d`	enable debug message
`-i`	start up in DLite! debugger
`-q`	terminate immediately
`-seed <num>`	random number generator seed
`-chkpt <fname>`	restore EIO trace execution
`-redir:sim <fname>`	redirect simulator output
`-redir:prog <fname>`	redirect simulated program output
`-nice <num>`	priority of simulator
`-max:inst <num>`	max number of instructions to execute

Miscellaneous Options

`-v`	verbose operation, prints instruction trace
`-vr`	prints register file after every instruction (-v must be specified)
`-trigger:inst <num>`	instruction number to begin verbose operation (-v must be specified)
`-fastfwd <num>`	skips the first <num> instructions, then begins the timing simulation

Note: For more information on using instruction tracing to debug problems, see the file doc/tracing.readme

Configuration Files

- Configuration commands:

  `-config <file>` - read configuration parameters from `<file>`

  `-dumpconfig <file>` - save configuration parameters into `<file>`

- Generating a configuration file:
  - specify non-default options on command line
  - include “`-dumpconfig <file>`” to generate configuration file
- Comments allowed in configuration files:
  - text after “#” ignored until end of line
- Configuration files may reference other configuration files
- Sample configuration files available in the `config` directory

Fetch Parameters

`-fetch:speed <num>` (default: 1)

Speed of front end relative to the core. If the decode width is 4 and the fetch speed is 3, (up to) 12 instructions will be fetched each cycle.

`-fetch:mult_lines <true | false>` (default: false)

Allows instructions to be fetched from multiple lines in the same cycle. The default behavior is false. This option is included to be compatible with sim-outorder v3.0.

`-fetch:mf_compat <true | false>` (default: false)

Turns on optimistic misfetch recovery. The default behavior is off. This option is included to be compatible with sim-outorder v3.0.

Fetch Parameters (cont.)

`-ifq:size <num>` (default: 32)

Size of the instruction fetch queue (IFQ) between the fetch and dispatch stages.

`-ifq:delay <num>` (default: 3)

Number of cycles an instruction must wait in the IFQ before being dispatch. This parameter replaces the –bpred:mplat and can be used to add more stages to the front-end pipeline.

Note: For compatibly, the IFQ size in MASE should be greater than the IFQ size in sim-outorder by (ifq:delay * decode:width * fetch:speed).

Specifying the Branch Predictor

- Specifying the branch predictor type:

```
-bpred <type>
```

where <type> is:

`nottaken`	always predict not taken
`taken`	always predict taken
`perfect`	perfect predictor
`bimod`	bimodal predictor (BTB w/ 2 bit counters)
`2lev`	2-level adaptive predictor
`comb`	combination of bimodal and 2-level predictor

- Configuring the bimodal predictor:
  (only useful when “`-bpred bimod`” or “`-bpred comb`” is specified)

`-bpred:bimod <size>`	size of direct-mapped BTB

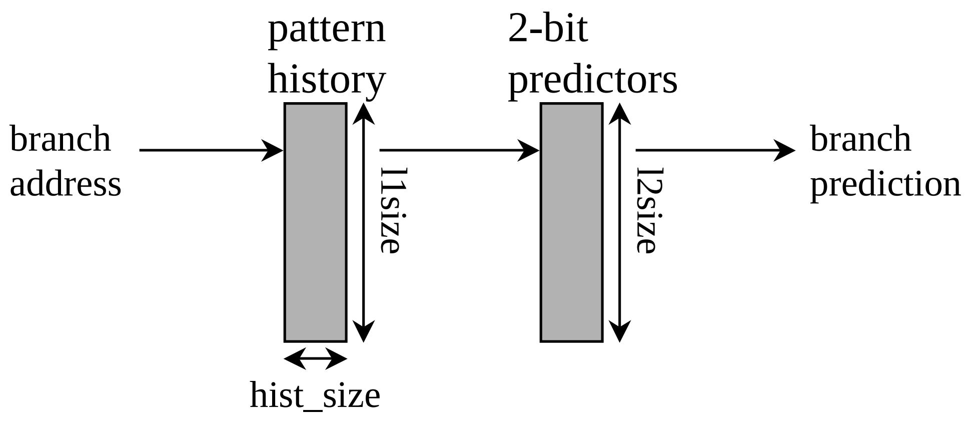
Specifying the Branch Predictor (cont.)

- Configuring the 2-level adaptive predictor:
  (only useful when "`-bpred 2lev`" or "`-bpred comb`" is specified)

```
-bpred:2lev <l1size> <l2size> <hist_size>
```

where:

`<l1size>`	size of the first level table
`<l2size>`	size of the second level table
`<hist_size>`	history (pattern) width



Specifying the Branch Predictor (cont.)

- Configuring the combining predictor:
  (only useful when "`-bpred comb`" is specified)

`-bpred:comb <size>`	size of meta table

- Other branch predictor options:

`-bpred:ras <num>`	size of return address stack
`-bpred:btb <sets> <assoc>`	BTB config, specifies number of sets and assoc.
`-bpred:spec_update <ID|WB>`	speculatively update branch predictor at decode (ID) or writeback (WB), default is to update at commit (CT).
`-bpred:rcvr_shadow <t|f>`	allows recovery in the shadow of another branch misprediction, default is true.

Dispatch / Issue / Commit Parameters

`-decode:width <num>` (default: 4)

Maximum number of instructions that can be decoded and dispatched per cycle.

`-issue:width <num>` (default: 4)

Maximum number of instructions that can be issued per cycle.

`-issue:inorder <true|false>` (default: false)

Runs the pipeline with in-order issue.

`-issue:wrongpath <true|false>` (default: true)

Issues speculative instructions (instructions in the shadow of a mispredicted branch) when set.

`-commit:width <num>` (default: 4)

Maximum number of instructions that can be committed per cycle.

Register Scheduler Parameters

`-rob:size <num>` (default: 16)

Number of entries in the reorder buffer (ROB).

`-rs:size <num>` (default: 16)

Number of reservation stations.

`-scheduler:delay <num>` (default: 0)

Number of cycles instruction must wait in the scheduler queue before starting to execute.

`-scheduler:replay <true|false>` (default: false)

Turns on latency prediction. Instructions are replayed if the prediction is too small.

Memory Scheduler Parameters

`-lsq:size <num>` (default: 8)

Number of entries in the load store queue (LSQ).

`-lsq:perfect <true|false>` (default: false)

Turns on perfect memory disambiguation. Allows loads to execute as soon as its dependencies are satisfied. Stores with unknown addresses do not block unless they write to the same address as the load.

`-lsq:blind <true|false>` (default: false)

Turns on blind load speculation. Allows loads to execute as soon as its dependencies are satisfied. If an earlier store with an unknown address writes to the address of the load, a blind load misprediction occurs.

Specifying Cache and TLB Configurations

- All caches and TLB configurations are specified with this format:

```
<name>:<nsets>:<bsize>:<assoc>:<repl>
```

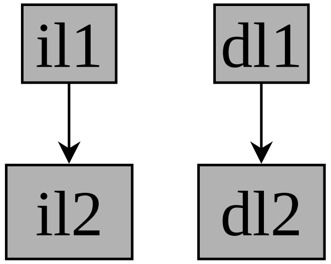
`<name>`	- cache name (make this unique)
`<nsets>`	- number of sets
`<bsize>`	- block size
`<assoc>`	- associativity (number of “ways”)
`<repl>`	- set replacement policy:
	`l` - for LRU
	`f` - for FIFO
	`r` - for RANDOM

- Examples:

`il1:1024:32:2:l`	2-way set-assoc 64k-byte cache, LRU
`dtlb:1:4096:64:r`	64-entry fully assoc TLB w/ 4k pages, random replacement

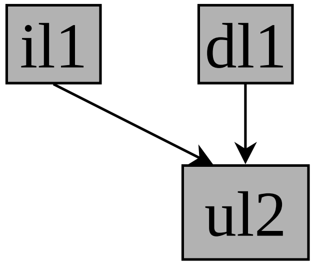
Specifying Cache Hierarchies

- You can specify all cache parameters such that no unified levels exist:



```
-cache:il1 il1:128:64:1:l
-cache:il2 il2:128:64:4:l
-cache:dl1 dl1:256:32:1:l
-cache:dl2 dl2:1024:64:2:l
```

- You can unify any level of the hierarchy, by “pointing” an I-cache level into the D-cache hierarchy:



```
-cache:il1 il1:128:64:1:l
-cache:il2 dl2
-cache:dl1 dl1:256:32:1:l
-cache:dl2 ul2:1024:64:2:l
```

Summary of Cache / TLB / Memory Parameters

`-cache:dl1 <config>`	- level 1 data cache configuration
`-cache:dl1lat <cycles>`	- level 1 data cache hit latency
`-cache:dl2 <config>`	- level 2 data cache configuration
`-cache:dl2lat <cycles>`	- level 2 data cache hit latency
`-cache:il1 <config>`	- level 1 instruction cache configuration
`-cache:il1lat <cycles>`	- level 1 instruction cache hit latency
`-cache:il2 <config>`	- level 2 instruction cache configuration
`-cache:il2lat <cycles>`	- level 2 instruction cache hit latency
`-cache:flush`	- flush all caches on system calls
`-cache:icompress`	- remap 64-bit inst addr. to 32-bit equiv.
`-tlb:itlb <config>`	- instruction TLB configuration
`-tlb:dtlb <config>`	- data TLB configuration
`-tlb:lat <cycles>`	- latency to service a TLB miss
`-mem:lat <1st> <next>`	- memory access latency (first, rest)
`-mem:width`	- width of memory bus (in bytes)

Resource Parameters

`-res:ialu`	number of integer ALUs
`-res:imult`	number of integer multiplier/dividers
`-res:memports`	number of first-level cache ports
`-res:fpalu`	number of FP ALUs
`-res:fpmult`	number of FP multiplier/dividers

Resource Parameters (cont.)

- MASE allows the latencies of the functional units to be specified. For each type of functional unit, there is an `oplat` (operation latency) and `islat` (issue latency). `oplat` refers to how long an instruction takes to execute. `islat` refers to how long it takes before the functional unit can initiate another request.

- Examples:

`-res:fmul_oplat 4`
`-res:fmul_islat 1`
	FP multiplies take 4 cycles to execute and a new request can be initiated each cycle.

`-res:fdiv_oplat 12`
`-res:fdiv_oplat 9`
	FP divideds take 12 cycles to execute and a new request can be initiated after 9 cycles.

Checker Parameters

`-chk:inject_err <true|false>` (default: false)

Randomly injects errors into core results. Can be used to verify if the checker and recovery are working correctly. The type of error can be changed by the programmer to anything they want in mase-commit.c.

`-chk:print_err <true|false>` (default: false)

Prints error messages associated with checker errors.

`-chk:stop_on_err <true|false>` (default: false)

Aborts the simulation when encountering a checker error.

`-chk:threshold <num>` (default: 1.000)

Specifies a threshold of num %. If the percentage of checker errors exceeds the threshold, a warning message is printed. This can be used to make sure the number of checker errors does not impact the performance of the program.

PC-Based Statistical Profiles

- Produces a text segment profile for any integer statistical counter.
- Specify a statistical counter to be monitored using "-pcstat" option. Works on any integer counter registered with the stats package, including those added by users.
- Example applications:

`-pcstat sim_num_insn`	- execution profile
`-pcstat sim_num_refs`	- reference profile
`-pcstat il1.misses`	- L1 I-cache miss profile
`-pcstat bpred_bimod.misses`	- br pred miss profile

- View with the `textprof.pl` Perl script, it displays pc-based statistics with program disassembly:

```
textprof.pl <dis_file> <sim_output> <stat_name>
```

PC-Based Statistical Profiles (cont.)

- Example usage:

```
sim-profile -pcstat sim_num_insn test-math >&! test-
math.out
objdump -dl test-math >! test-math.dis
textprof.pl test-math.dis test-math.out sim_num_insn_by_pc
```

- Example output:

```
        00401a10:  ( 13,   0.01): <strtod+220> addiu $a1[5],$zero[0],1
        strtod.c:79
        00401a18:  ( 13,   0.01): <strtod+228> bc1f 00401a30
<strtod+240>
        strtod.c:87
        00401a20:                : <strtod+230> addiu $s1[17],$s1[17],1
        00401a28:                : <strtod+238> j 00401a58 <strtod+268>
        strtod.c:89
        00401a30:  ( 13,   0.01): <strtod+240> mul.d $f2,$f20,$f4
        00401a38:  ( 13,   0.01): <strtod+248> addiu $v0[2],$v1[3],-48
        00401a40:  ( 13,   0.01): <strtod+250> mtc1 $v0[2],$f0
```

executed 13 times

never executed

Pipetraces

- Pipetrace produces detailed history of all instructions executed, including instruction fetch, retirement, and stage transitions.
- Use the "-ptrace" option to generate a pipetrace:

```
-ptrace <file> <range>
```

- Example usage:

`-ptrace FOO.trc :`	- trace entire execution to FOO.trc
`-ptrace BAR.trc 100:5000`	- trace from inst 100 to 5000
`-ptrace UXXE.trc :10000`	- trace until instruction 10000

- Consult the "Execution Ranges" section to learn what can be specified using the <range> parameter.
- View pipetrace file with the `pipeview.pl` Perl script. It displays the pipeline for each cycle of execution traced:

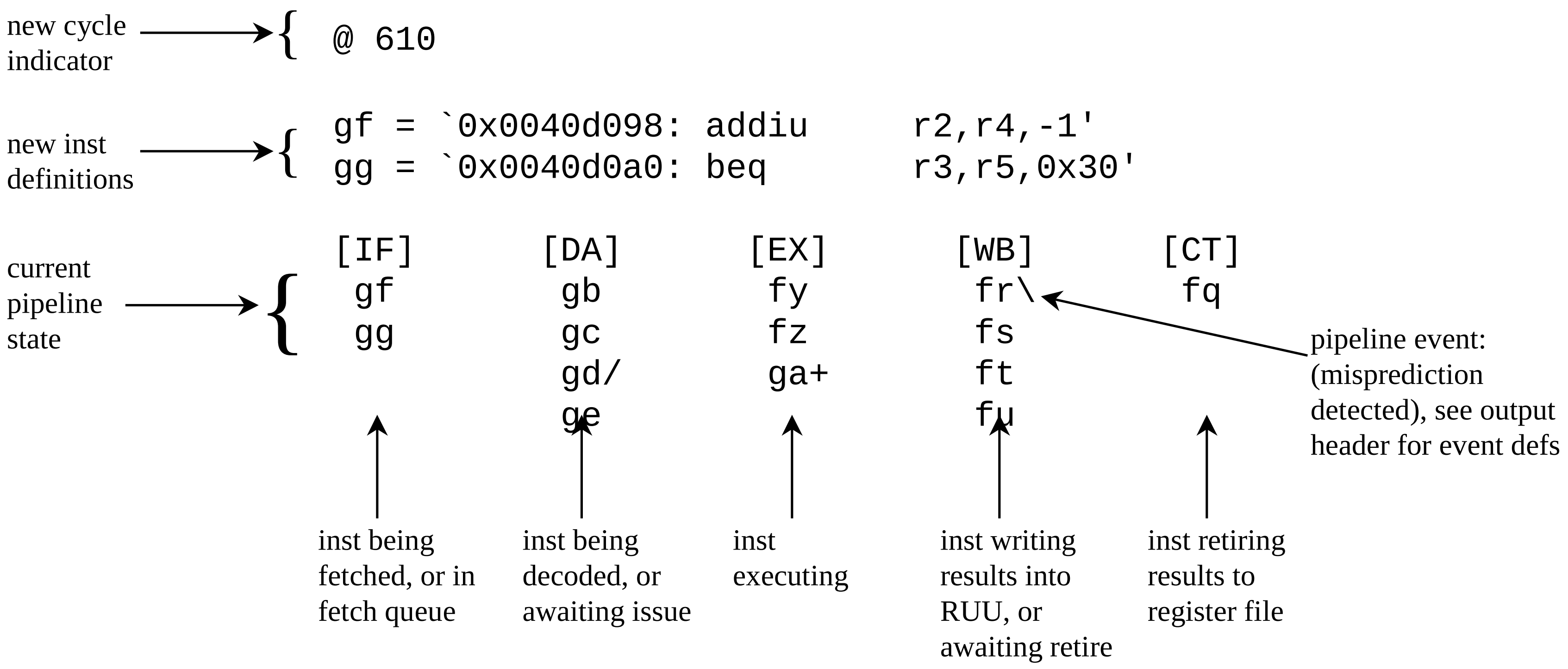
```
pipeview.pl <ptrace_file>
```

Pipetraces (cont.)

- Example usage:

```
sim-outorder -ptrace FOO.trc :1000 test-math
pipeview.pl FOO.trc
```

- Example output:



```
@ 610

gf = `0x0040d098: addiu     r2,r4,-1'
gg = `0x0040d0a0: beq       r3,r5,0x30'

[IF]      [DA]      [EX]      [WB]      [CT]
 gf        gb        fy        fr\       fq
 gg        gc        fz        fs
           gd/       ga+       ft
           ge                  fu
```

new cycle indicator
new inst definitions
current pipeline state
pipeline event: (misprediction detected), see output header for event defs
inst being fetched, or in fetch queue
inst being decoded, or awaiting issue
inst executing
inst writing results into RUU, or awaiting retire
inst retiring results to register file

Part 4: Additional Topics

DLite!, the Lite Debugger

- DLite! is a very lightweight symbolic debugger that can be activated with the "`-i`" option (interactive).
- In MASE, the debugger resides in commit. A step will stop at the retirement of the next retired instruction or the next cycle, whatever comes first.
- Program symbols and expressions may be used in most contexts, for example: "`break main+8`"
- Many structures specific to MASE have dump routines. See "`help mstate`".
- Use the "`help`" command for complete documentation.
- Main features of DLite!:

– `break`, `dbreak`, `rbreak`:	set text, data, and range breakpoints
– `regs`, `iregs`, `fregs`:	display all, int, and FP register state
– `dump <addr> <count>`:	dump `<count>` bytes of memory at `<addr>`
– `dis <addr> <count>`:	disassemble `<count>` insts starting at `<addr>`
– `mstate`:	display machine-specific state
– `print <expr>`, `display <expr>`:	display expression or memory

DLite!, the Lite Debugger (cont.)

- Breakpoints:
  - code:
    - `break <addr>`
    - `Examples: break main`, `break 0x400148`
  - code:
    - `rbreak <range>`

    where range is a range (see Execution Ranges section)
    - `Examples: rbreak @main:+279, rbreak 2000:3500`
  - data:
    - `dbreak <addr> {r|w|x}`

    where r == read, w == write, x == execute
    - `Examples: dbreak stdin w`, `dbreak sys_count wr`
- DLite! expressions:

  operators: +, -, /, *

  literals: 10, 0xff, 077

  symbols: main, vfprintf

  registers: $r1, $f4, $pc, $fcc, $hi, $lo

Execution Ranges

- specify a range of addresses, instructions, or cycles
- used by range breakpoints and pipetrace
- format:

address range:	@<start>:<end>
instruction range:	<start>:<end>
cycle range:	#<start>:<end>

- the end range may be specified relative to the start range
- both endpoints are optional, and if omitted the value will default to the largest/smallest allowed value in that range
- e.g.,
  - @main:+278- main to main+278
  - #:1000 - cycle 0 to cycle 1000
  - : - entire execution (instruction 0 to end)

Additional Information

- MASE is part of SimpleScalar release 4.0. Source code is available for non-commercial use at:

  `http://www.simplescalar.com`

- The above website also contains documentation, related tools, and information on SimpleScalar mailing lists.

- ISPASS paper:

  Eric Larson, Saugata Chatterjee, and Todd Austin, "MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling", 2001 IEEE International Symposium on Performance Analysis of Systems and Software, November 2001.