

1 *sim-alpha* README file

1.1 Installing *sim-alpha*

sim-alpha currently builds only on x86/linux boxes. The simulator comes equipped with its own loader. To build the simulator, uncompress the *tgz* file and type *make* in the resulting *alphasim* directory. This should build the simulator *sim-alpha*. The *alphasim/tests* directory contains compiled test binaries. The simulator will run any binary compiled for the alpha ISA. *sim-alpha* takes command line arguments and also accepts arguments in a file. Use the *-config* option to specify a configuration file. The simulator can be compiled in three modes:

1. Normal mode where it includes all Alpha 21264 features
2. Flexible mode where the low level features in the 21264 can be turned on or off. Define *FLEXIBLE_SIM* during compilation to run in this mode.
3. Functional debug mode where a functional simulator checks the correctness of the timing simulator. Enable *FUNC_DEBUG* flag during compilation for this version of the simulator. While running in functional debug mode, early instruction retire should be disabled, and only *eio* traces should be used.

1.2 Using *sim-alpha*

Below we list some of the options found in *sim-alpha* along with a brief description:

General simulator configuration:

-config - Configuration file to use

- chkpt** - Check point file to start execution from for eio traces
- max:inst** - Maximum number of committed instructions to simulate
- fastfwd** - Number of instructions to fast forward before starting timing simulation. Note the caches are not kept warm during this phase.

Processor core configuration

- mach:freq** - Frequency of simulated machine. This is used by system calls like rusage.
- fetch:ifqsize** - Fetch queue size. The fetch stage stalls if the fetch queue cannot accommodate fetch width number of instructions
- fetch:speed** - Number of discontinuous fetch width fetches per cycles
- fetch:width** - Number of instructions to fetch each cycle
- slot:width** - Number of instructions which can be slotted per cycle
- map:width** - Number of instructions which can be mapped per cycle
- issue:intwidth** - Number of integer instructions which can be issued per cycle
- issue:fpwidth** - Number of fp instructions which can be issued per cycle
- commit:width** - Number of instructions which can be committed per cycle
- fetch:stwait** - Species size of the stWait table
- issue:int_reg_lat** - Integer register read latency
- issue:fp_reg_lat** - fp register read latency
- issue:int_size** - Integer issue queue size
- issue:fp_size** - fp issue queue size
- reg:int_p_regs** - Number of integer physical registers
- reg:fp_p_regs** - Number of fp physical registers
- rbuf:size** - Species the size of the reorder buffer

- lq:size** - Specifies the size of the load queue
- sq:size** - Specifies the size of the store queue
- res:ialu** - Number of integer ALU units
- res:imult** - Number of integer multiplier units
- res:fpalu** - Number of floating point ALU units
- res:fpmult** - Number of floating point multiplier units
- res:iclus** - Number of integer clusters
- res:fpclus** - Number of fp clusters
- res:delay** - Minimum cross cluster delay between adjacent clusters

Memory Hierarchy flags

- bus:queuing_delay** - Enables delay due to contention in the buses to lower levels of the memory hierarchy
- cache:perfectl2** - Simulates perfect level 2 cache
- prefetch:dist** - Number of blocks to prefetch on a level 1 I-cache miss
- cache:define** - *< name >: < nsets >: < bsize >: < subblock >: < asso >: < repl >: < lat >: < trans >: < #resources >: < rescode >*

where

- < name >* - Defines name of the cache
- < nsets >* - Specifies the number of sets in the cache
- < bsize >* - Specifies the block size of the cache
- < subblock >* - Specifies sub-block size of the cache
- < asso >* - Specifies associativity of the cache
- < repl >* - Species the cache replacement policy. Choose from LRU, FIFO, random, and LFU

< *lat* > - Hit latency

< *trans* > - Translation policy (vibt, vipt, pipt)

-cache:dcache - defines name of first level data cache

-cache:icache - defines name of first level instruction cache

-cache:vbuf_lat - Victim buffer latency

-cache:vbuf_ent - Number of entries in the victim buffer

-cache:mshrs - Sets maximum number of MSHRs per cache

-cache:prefetch_mshrs - Sets maximum number of prefetch MSHRs per cache

-cache:mshr_targets - Sets maximum number of allowable targets per mshr

-bus:define - < *name* >:< *width* >:< *cyclelatency* >:< *arbitrationpenalty* >:< *infbandwidth* >:< *#resources* >:< *resourcecode* >:< *resourcenames* >

where

< *name* > - Specifies bus name

< *width* > - Specifies bus width in bytes

< *cyclelatency* > - bus latency in terms of processor cycles

< *arbitrationpenalty* > - Number of cycles for arbitration

< *infbandwidth* > - if 1, infinite bandwidth

< *resourcenames* > - Name of structure to which this bus connects (eg. L2)

-mem:clock_multiplier - ratio of cpu frequency to DRAM frequency

-mem:page_policy - DRAM page policy (0 - open page, 1 - close page autoprecharge)

-mem:ras_delay - time between start of ras command and cas command

-mem:cas_delay - time between start of cas command and data start

-mem:pre_delay - time between start of precharge command and ras command

-mem:data_rate - 1 specifies single data rate and 2 specifies double data rate

-mem:bus_width - width of bus from cpu to dram

-mem:chipset_delay_req - delay in chipset for request path

-mem:chipset_delay_return -delay in chipset in data return path

-tlb:define - *< name >: < nsets >: < bsize >: < subblock >: < assoc >: < repl >: < hitlatency >: < translation >: < prefetch >: < #resources >: < resourcecode >*

the flags have similar meanings to cache define.

-tlb:itlb - Name of instruction TLB

-tlb:dtlb - Name of data TLB

Predictor configuration Line and way predictor

-line_pred:ini_value - Initial value of the 2-bit line predictor training counter

-line_pred:width - Number of instructions for which the line predictor stores a prediction

-way:pred - Specifies the latency of the way predictor

Branch predictor

The branch predictor configuration supports the following new flags along with those specified in the SimpleScalar 2.0 release documentation.

-bpred *< type >*

21264 21264 tournament predictor

The predictor-specific arguments are listed below:

-bpred:21264 - *< l1size > < l2size > < lhist_size > < gsize > < ghist_size > < csize > < chist_size >*

where the different flags are

- `< l1size >` - the local predictor level 1 table size
- `< l2size >` - the local predictor level 2 table size
- `< lhist_hist >` - the number of history bits in the second level table
- `< gsize >` - the size of global predictor
- `< ghist_size >` - number of history bits in the global predictor
- `< csize >` - choice predictor size
- `< chist_size >` - number of history bits in the choice predictor

Low level feature configuration For all these features, 1 enables them and 0 disables them. These features can be made operational by compiling the simulator with the FLEXIBLE_SIM flag.

- issue:no_slot_clus** - 1 disables clustering of functional units
- slot:adder** - Enable/disable the adder for computing targets of PC relative control instructions in the slot stage
- slot:slotting** - Enable/disable static slotting in the slot stage
- map:early_retire** - Enable/disable early instruction retire in the map stage
- wb:load_trap** - Enable/disable load traps
- wb:diffsize_trap** - Enable/disable traps due to loads and stores with different sizes to the same address
- cache:target_trap** - Enable/disable trap if two loads map to same MSHR target
- cache:mshrfull_trap** - Enable/disable trap if MSHRs are full
- map:stall** - Enable/disable stall of map stage for 3 cycles if the number of free physical registers to map falls below 8
- bpred:spec_update** - Enables speculative update of the global and choice predictors if 1
- line_pred:spec_update** - Enables speculative update of line predictor if 1

-load:spec - Enable/disable load use speculation

1.3 *sim-alpha* internals

In this section, we describe the internal working of *sim-alpha*. *sim-alpha* is an execution-driven simulator and it executes instructions down the mis-speculated path, in the same way an actual processor would execute them. Thus, *sim-alpha* can capture the behavior of these mis-speculated instructions. However, a disadvantage of this approach is that the correct path is known only at commit time, and hence perfect prediction cannot be simulated. Each stage of the 21264 pipeline is modeled in a separate file, except the register read stage, which is bundled along with issue stage. The main loop of the simulator, located in `simulate.c`, looks as follows:

```
while(TRUE) {
    eventq_service_events();
    commit_stage();
    writeback_stage();
    issue_stage();
    map_stage();
    slot_stage();
    fetch_stage();
}
```

This loop is walked once for each simulated machine cycle. The simulation is partly event driven. Before the entering the main loop, the simulator calls the initialization functions for each pipeline stage. These functions set up the various data structures needed for timing simulation. If fast-forwarding is enabled, functional simulation is performed till the required number of instructions are committed, and then timing simulation starts.

The simulator also supports eio tracing. eio traces ensure that the values returned by system calls are the same across different runs. With eio traces, the user can also drop checkpoints, and later start simulation from the check point.

The fetch stage of the pipeline is implemented in *fetch_stage()*. The fetch stage accesses the instruction cache, and fetches *fetch_width* number of instructions per access. The accesses are always aligned on a *fetch_width* boundary. The fetch speed determines how many discontinuous fetches can be performed by the processor per cycle. The line predictor provides the next I-cache line to fetch.

The slot stage of the pipeline is implemented in *slot_stage()*. In the slot stage, instructions are statically slotted to either the UPPER or the LOWER sub-clusters depending on their position in the fetch packet, and their resource requirements. The slotting algorithm is implemented as an array *slot_instclass2slotclass*. The control instructions also access the branch predictor in this function with a call to *bred_lookup()*. For PC relative control instructions, the target is computed using an adder. The speculative update of the branch and the line predictors also happens in this stage. For certain classes of instructions, the slot stage overrides the line predictor prediction with the branch predictor prediction if they differ. In this case, the remaining instructions in the fetch queue are squashed, and fetch is restarted the next cycle by changing *regs_PC* to the branch predictor target address. We also check to make sure that the fetch is continuous in the absence of control instructions. If not, the fetch is restarted with the correct PC. However, we don't charge any penalty here as the 21264 fetch engine can detect the absence of control instructions the same cycle.

The map stage is implemented in the function *map_stage()* in *map.c*. The map stage initially identifies the input and output registers for the current instruction. It then checks for the availability of a reorder buffer entry, integer or fp issue queue entry depending on whether the instruction has an integer or fp destination register, output physical register, and load or store queue entry if the instructions is a load or store instruction. The map stage then identifies the input physical physical registers for this instruction, and if these are ready, puts the instruction in the ready queue. Else, the map stage identifies the producer(s) of the instructions operands, and queues the instruction for wakeup, by linking it to the dependence chain entry of the producers reorder buffer entry. Some instructions with destination register 31 are also retired in this stage as part of early instruction retire. FTOI and fp store instructions are queued in both integer and fp queues, and ITOF and fp load instructions are queued in the integer queue. The map stage also checks the *stWait* bit for load instructions, and if set, it adds these instructions to the dependence chain of the last store

instruction before the current load. CMOV instructions have three input operands, and these are handled using the *cmovdeps* dependence chain. The map stage also assigns a unique *inum* to each instruction, which is used for determining its age, for insertion into the ready queue.

The code for issue stage resides in the file *issue.c*, and is implemented in the function *issue_stage()*. The issue stage is responsible for picking instruction from the integer and floating point ready queues, checking for the availability of functional unit, and issuing the instruction to the functional unit. The register read latency is also charged here. Instructions whose operands are ready are inserted in the integer and fp ready queues by the *writeback_stage()* and the *map_stage()*. These instructions also have corresponding entries in the integer and fp issue queues. Instructions are maintained in the ready queue strictly by age. Each cycle, the issue stage takes integer and fp issue width number of oldest instructions from the ready queues, and issues them, provided these instructions satisfy their functional unit requirements. The issue stage uses the slotting performed by the slot stage in deciding which sub-cluster to issue an instruction. The functional unit code is contained in *resource.c*. The function *res_find_clus()* finds the cluster on which an instruction can execute. The functional unit allocation is rather hard coded in *sim-alpha* because of the presence of clusters and sub-clusters in the 21264. By destroying clustering, the functional unit code can be made more scalable. The issue stage also assigns the appropriate latency for each instruction depending on its type. The issue stage finally sets up events to free the instruction entry in the integer and fp queues 2 cycles after issue, and to signal completion of execution of this instruction.

The writeback stage implemented in *writeback_stage()* is responsible for waking up the dependent instructions when a producing instruction completes. In the simulator, the functional execution of the instruction also takes place in this stage. The writeback stage picks up completed instructions from the event queue, and walks their dependent chain marking the operands of dependent instructions ready. If all the operands of an instructions are ready, it inserts these instructions in the ready queue. The writeback stage also sets the reorder buffer completed entry for these instructions. The target for control instructions are resolved, and mis-predictions are indicated in the corresponding reorder buffer entry, along with the correct target. Load instructions access the D-cache, and store instructions are marked as ready to access the D-cache during commit.

The commit stage in *commit_stage()* is responsible for retiring instructions from the reorder buffer. Every cycle, the commit stage retires *commit_width* number of instructions. However, the commit stage cannot retire past a branch. The commit stage examines the head of the reorder for mis-predictions and traps, and flushes the pipeline by calling *commit_flush_pipeline()* if the instruction caused a mis-predict or a trap. Otherwise, it retires the instruction, and updates the architectural register file. The commit stage also sends the store instructions to D-cache.

sim-alpha incorporates a detailed memory system with support for multi-level cache hierarchy, address translations, bus contention, and SDRAM memory model. The memory code is mostly event driven, and is scheduled by the global event queue contained in *eventq.c*. At the beginning of each cycle, the *eventq_service_events()* function checks for memory operations completing that cycle. The files *cache_timing.c*, *tlb.c*, *bus.c*, *memory.c*, and *dram.c* contain most of the memory timing code for *sim-alpha*.