Timing Model of Superscalar O-o-O processor in HAsim Framework

Submitted by

Muralidaran Vijayaraghavan

I Introduction to Timing Models:

Software simulation of micro-architecture designs before actually developing the complete synthesizable RTL for the design is the standard practice to evaluate microprocessor designs. This has a big advantage. Architects do not have to wait for the complete design of the micro-architecture in RTL before having to evaluate the performance of a design. The performance of the entire design can be altered by tweaking a few parameters which control the timing information of the modules in the simulation, rather than having to worry about how that module can be designed to meet the expected timing. This helps the architect focus on the big picture rather than to worry about the exact details of each internal module in the micro-architecture design. This mode of simulation will be facilitated by partitioning the simulator into a timing partition and a functional partition. Figure 1 shows the partition of a simulator into its timing and functional partitions. In the software simulator, the functional partition can be written simply as a giant case statement, which maps each instruction type in the ISA into the corresponding change of architectural states of the processor. This is essentially a 1-cycle processor in some sense. Now, the timing partition drives instructions into this functional partition. The timing partition traditionally maintains all the micro-architectural states of the processor. It determines if the micro-architecture is going to be pipelined or not, the number of pipeline stages, whether the instructions can execute out-of-order or not, fetch width of the processor, commit width of the processor, number of parallel functional units, branch prediction algorithm used etc, to name a few. This difference between the timing partition and the functional partition can be summarized as shown in Figure 1 where Functional partition can be thought of as just implementing the ISA, whereas the timing partition implements the actual micro-architecture. Basically the timing model of a software simulator can be thought of as the control path in the actual processor design, except that it is written in sequential software, and hence is not going to execute concurrently. So the timing partition in software can take several cycles of CPU time in order to model one actual simulated time. So, normally there is some explicit mechanism to keep count of the actual simulated time, by incrementing a counter in the timing model each time a model cycle is finished. Also, the functional partition can take any number of cycles to implement the functionality of the instructions. The timing of the timing partition is decoupled from the timing of the functional partition. This helps in decoupling the timing partition completely from the functional partition. So the timing partition can implement a completely different micro-architecture while still retaining the same functional partition.

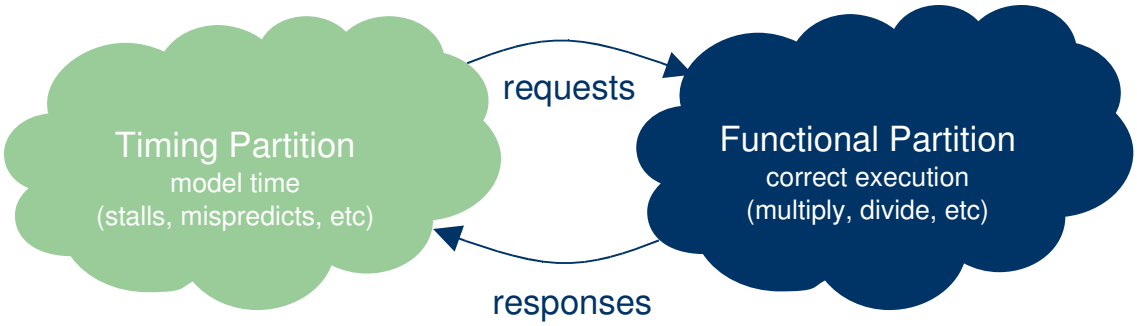

Figure 1

Timing and Functional partition in a simulator

II Introduction to HAsim

Classically, simulation models decrease accuracy and detail to improve simulation time. Models of large systems running large benchmark suites can result in simulation times of days, weeks, or months. At these speeds, simulation time dominates the overall cost of modeling, as shown in Figure 2.

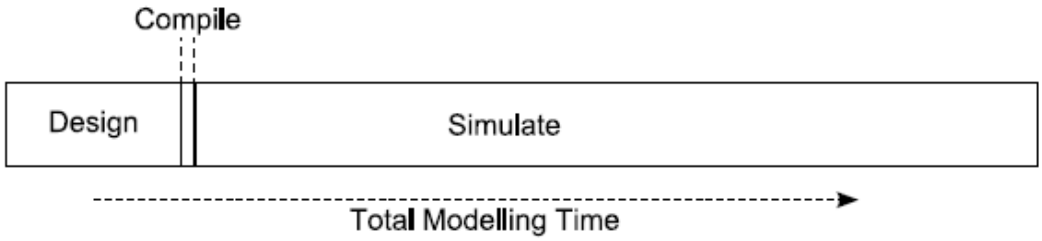

Figure 2
Software simulator design-compile-simulate cycle

In an effort to improve simulation speed without degrading accuracy, system architects are beginning to explore alternatives to software performance models. There can be considerable improvement in performance of models using FPGAs and high-level synthesis languages like Bluespec. Presumably this will lead to increased design time over software, and increased compilation time due to the complexity of FPGA synthesis

and placement tools. Ultimately, then our hope is to redistribute the cost of modeling as shown in Figure 3

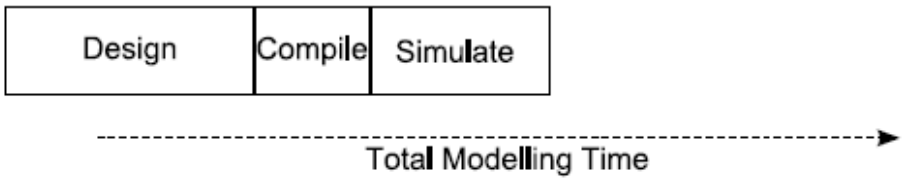

Figure 3
FPGA simulator design-compile-simulate cycle

HAsim, to be described in one sentence, is a framework to write software-like simulators, separating the timing partition and the functional partition as in software, but run the simulator in FPGAs. Michael Pellauer and Joel Emer are the chief architects behind HAsim. The main defect in software models is that software is inherently sequential, whereas the microprocessor, which it is trying to model is inherently parallel. Several operations execute in parallel in the microprocessor. Even in the case of a simple 5-stage MIPS pipeline, there are atleast 5 operations taking place in parallel in the 5-stage pipeline. So, even a dual core processor will not be able to sustain this level of parallelism exhibited by the simplest of the processors like the 5-stage MIPS processor.

But, if we use FPGAs, we can configure them to model the hardware directly, which will enable us to use the parallelism of hardware without any effort. I don't mean to implement the entire hardware in the FPGAs, but just to implement the software-simulator in the FPGAs. This is made possible only by the use of high level hardware description language like Bluespec, as designing the hardware to implement in the FPGAs using verilog can be compared to writing a software-simulator in assembly.

The following sections describe some of the concepts in HAsim: the design of the functional partition, the difference between an FPGA cycle and the model cycle, the concept of APorts, used to model ticks in the timing partition and finally the AWB GUI tool used to facilitate reuse of modules.

A Design of the functional partition

The functional partition, as I mentioned earlier, implements the functionality of the ISA. It executes the instructions one by one and maintains the architecturally visible states. Figure 4 shows the implementation of the functional partition. The green arrows coming out of the functional partition represent the interface of the functional partition with the timing partition. Each of the stages of the functional partition exports a server-like interface with the timing partition, ie each of the stages have a separate request and a

response port. Similarly, the token generator also has a request-response interface with the timing partition. Each stage of the functional partition also has a kill-request interface to which the timing partition sends requests for killing particular instructions. The functional partition maintains the architectural state of the whole system (processor + memory) using the RegState and the MemState. Figure 5 gives a more detailed description of the RegState. It consists of a physical register file (PRF), a mapping from the logical register to the physical registers, and an ROB to enable rollback of architectural states. Figure 5 also shows the presence of Checkpoints unit. This takes checkpoints of the architectural states at certain instances, and this helps in speeding up reloads (which would have proceeded at one instruction at a time using the ROB otherwise). Figure 6 shows the MemState of the Functional Partition. It has a store buffer, which stores all the stores till a global commit is requested by the timing partition. This store buffer is first checked to service any load request before checking the memory.

The instruction fetch cycle starts with the timing partition requesting a token from the functional partition. The token is the way of identifying a particular instruction at any stage in the functional partition. After receiving the token back from the functional partition, the timing model has to send the PC, and the token to the functional partition's fetch stage. The fetch stage in the functional partition associates the token with a particular instruction. The reply from the fetch stage has the token and the instruction it read from the memory. The instruction is read from the memory state (MemState) of the functional partition. The timing model then has to send a token for a decode request. The decode stage of the functional partition reads the RegState, to find the physical registers which this particular instruction depends on, and returns the set of these physical registers. Then the timing partition has to send an execute request. The functional partition returns an acknowledgement for this request. Then the timing partition has to send a local commit request followed by a global commit request, both of which are responded by acknowledgements by the functional partition. At every time instant, only one stage of the functional partition maintains information about the token, as an instruction can only be in one stage at a time. So whenever a kill request comes in from the timing partition to a particular stage in the functional partition, it clears the state for that instruction in that particular partition.

B FPGA cycle vs Model cycle

The number of cycles to perform a particular operation in real time is decoupled from the number of cycles the processor which we are modeling is going to take. That is the timing model maintains its own time, which is completely decoupled from the time that the functional partition takes to perform a particular function. Also, as mentioned before, the model doesn't have to exactly simulate the processor it is modeling. For example, the CAM structure is highly inefficient when modeling in FPGAs. But, if we have a mechanism to decouple the modeled time from the real time in the FPGAs, we can use a sequential lookup instead of a CAM structure.

C APorts – counting ticks in the timing partition

All communication between modules must occur through Asim ports or APorts ([1]). Asim ports also impose a discipline for modeling time. An Asim port is essentially a FIFO of type *t* with a user-specified latency *l* and bandwidth *b*. An Asim port has the following properties:

- A message travels through a port in *l* model clock cycles.
- At the beginning of every model cycle a module must read all of its input ports. It may then perform an arbitrary amount of calculation, and ends the cycle by writing all of its output ports.
- A module may send a message of type *t*, or a special value NoMessage (represented as an Invalid Maybe type in Bluespec), indicating no activity on this cycle.

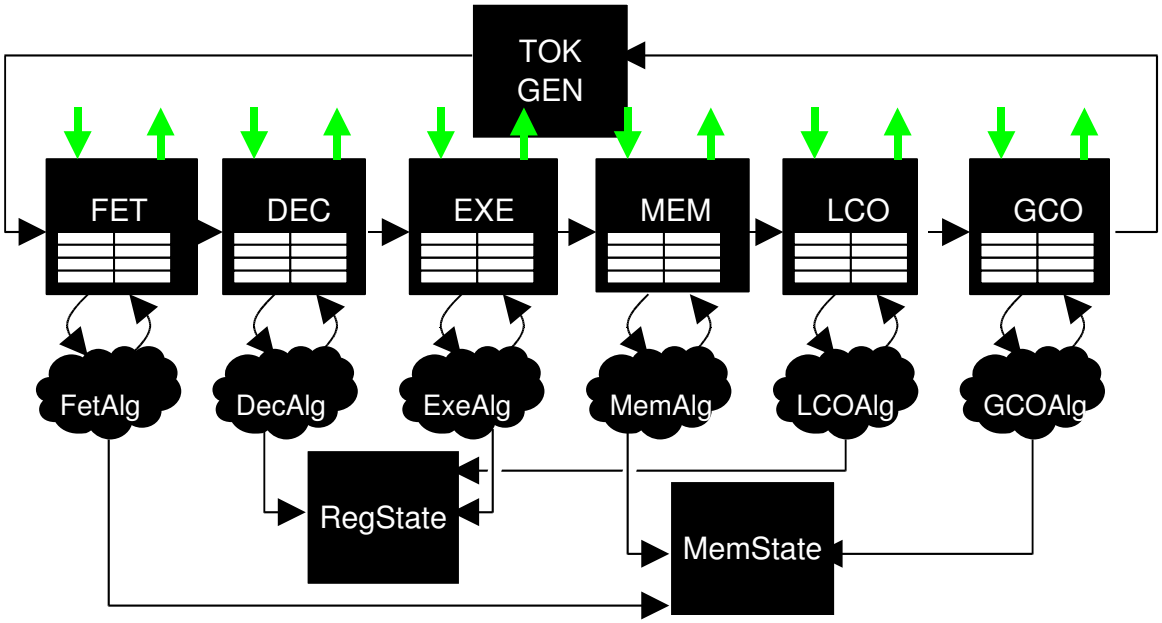

Figure 4
Functional Partition

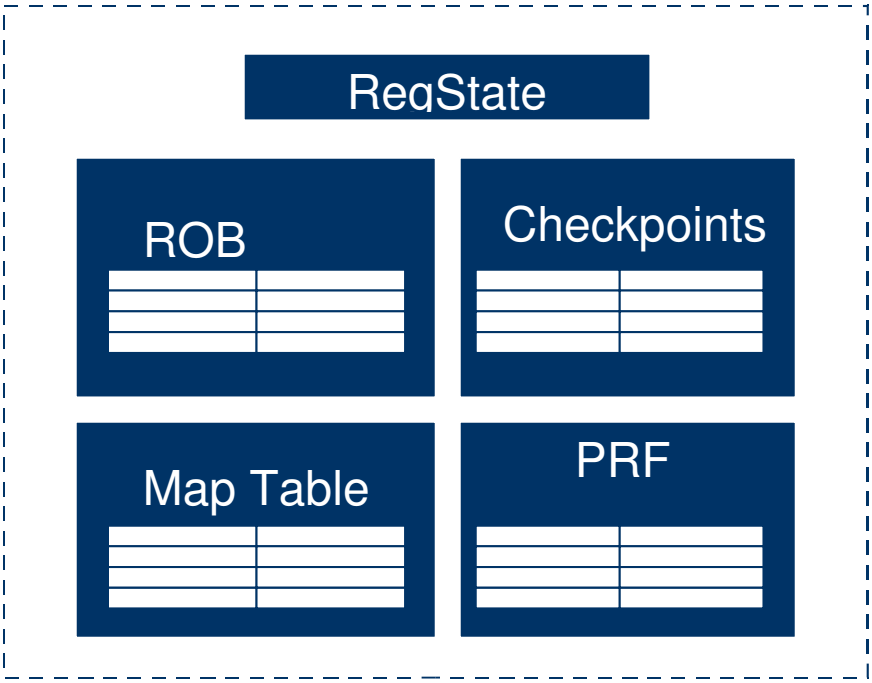

Figure 5
RegState of Functional Partition

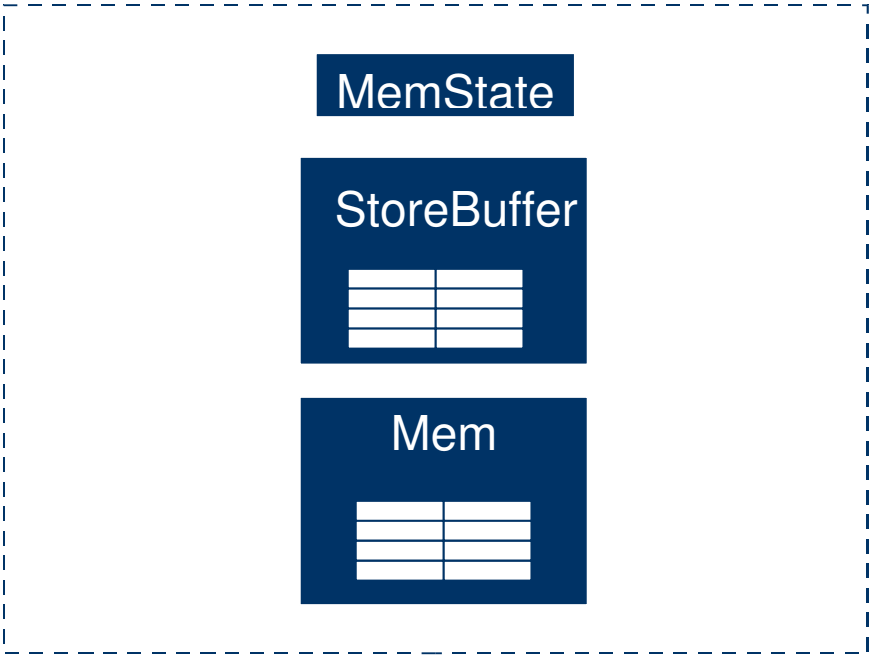

Figure 6
MemState of Functional Partition

The ASim abstraction offers several benefits to the model-writer. First, it allows the modules to be implemented separately, perhaps by separate programmers. Second, it eases verification, as programmers may be sure that there is no communication between modules except through Asim ports. Third, it enables certain types of design-space exploration by retiming operations via changing the Asim port latencies. Finally, it allows the model-writer to not worry about explicitly representing model time at all, but simply to communicate in a FIFO manner. This system of writing a timing module using A-Port is shown in Figure 7.

APorts do not restrict the modules to behave exactly in the way shown in Figure 7. It simply communicates the model time in a FIFO manner. So the ticks are more or less localized within each port of the module. This means that the module doesn't have to wait till it receives messages in all its input APorts before proceeding. If the module does not wait till it receives data from all the inputs, then we should change the definition of a model clock tick slightly. We can conceptually view a rising edge of a model clock to be when all the input APorts are read and a falling edge of a model clock to be when all the output APorts are written into (This is similar to the Bluespec clock cycle). The difference from the above definition is that there may not be any physical time between reading all input APorts and writing all output APorts, ie, some output APort can be written simultaneously as when some other input APort of the same module is read from. This abstraction speeds up simulation even more, as we don't have a barrier synchronization step where we have to wait to get data from all input APorts.

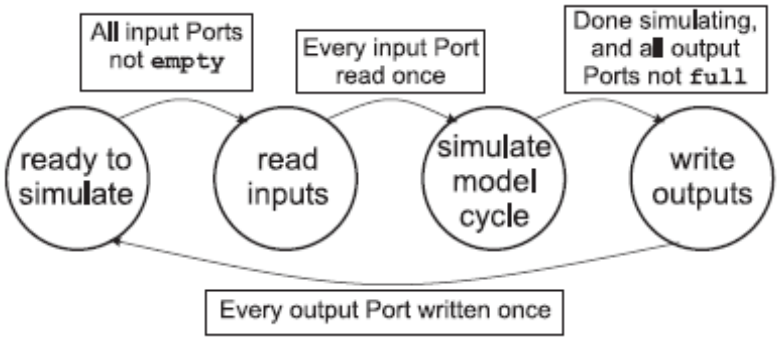

Figure 7
Interaction between module and ports

D Plug and Play using AWB

AWB or Asim Architect's Work Bench ([2]) used to be Intel's internal tool, which is now under GPL. This tool provides a graphical interface to plug and play different implementations of modules which provide the same interface. For example, different branch predictors which provide the same interface of getPredictedAddr() and updatePrediction(), can be implemented and all these different predictors will be displayed in the GUI. This enabled me to make incremental changes in the modules. For example, for the fetch unit, I had a module which fetched only 1 instruction at a time, then I had a module which fetched 4 instructions at a time. These have exactly the same interface, so they can be plugged in interchangeably. Using the first module creates a non-superscalar out of order processor, whereas the second module creates a superscalar out of order processor. Figure 8 shows a screenshot using AWB.

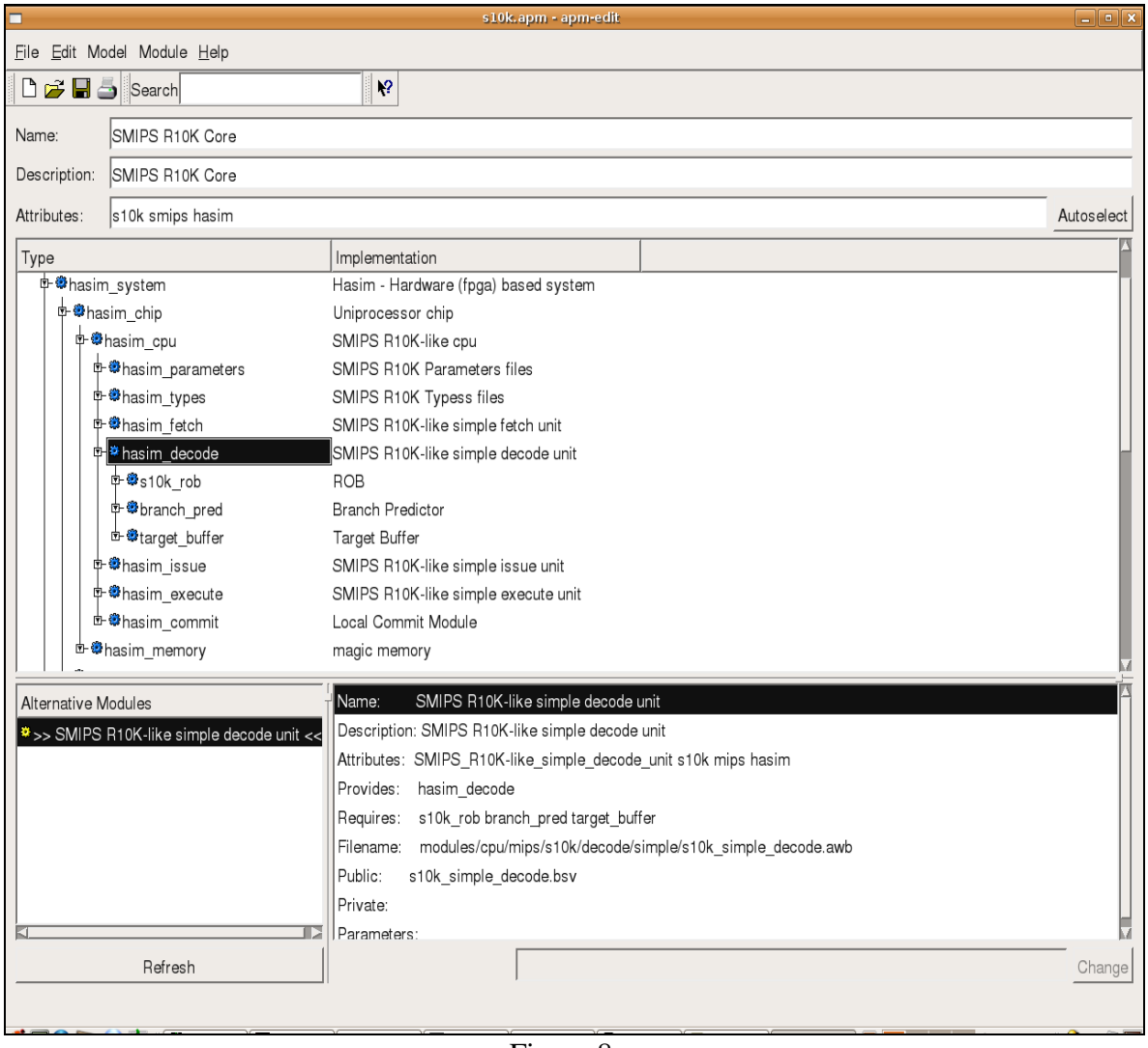

Figure 8

AWB GUI showing the R10k for SMIPS model

III Cycle accurate simulation model for R10k architecture using SMIPS ISA

My project was implementing a timing model for R10k architecture, but with the SMIPS ISA, in the HAsim infrastructure. Figure 9 gives the architecture of MIPS R10000, which is a superscalar out-of-order 64 bit machine. My timing model implementation is also for a superscalar out-of-order machine, but for SMIPS ISA which uses 32 bit registers and 32 bit addresses.

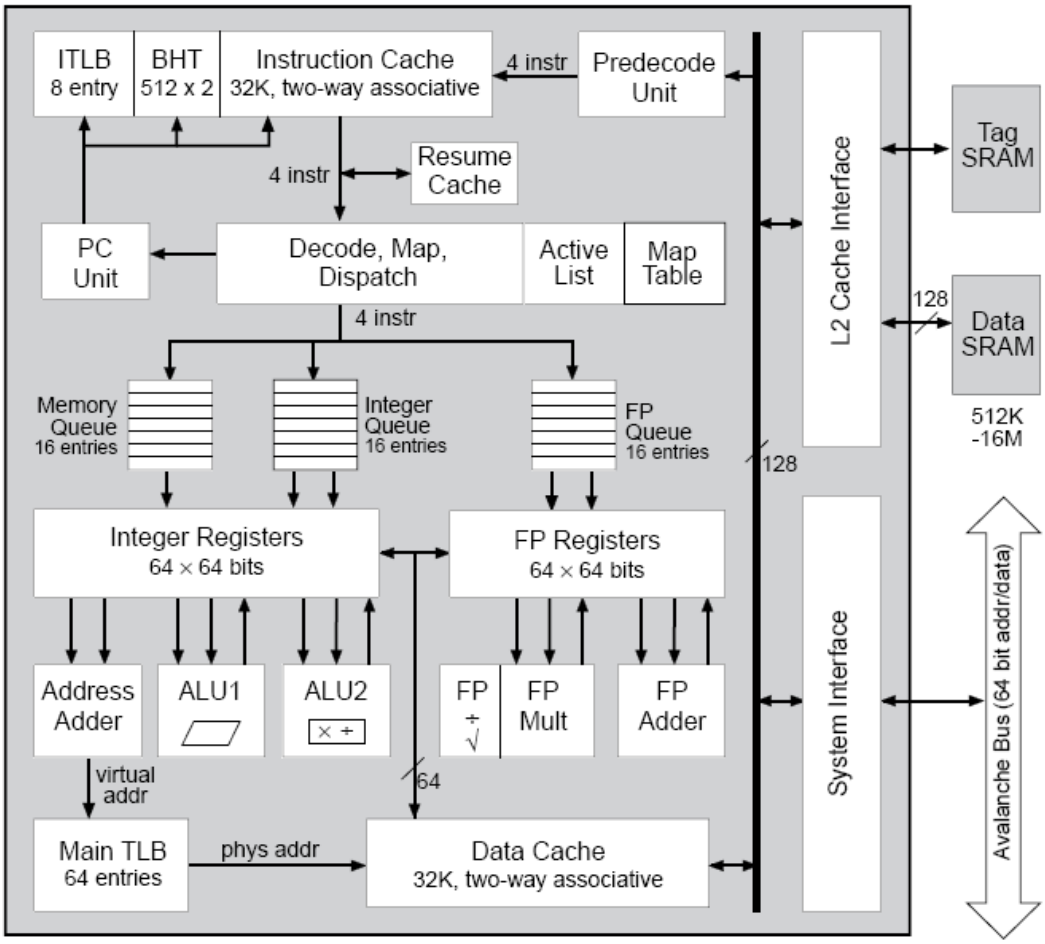

Figure 10
MIPS R100000 architecture

A MIPS R10000 architecture

This section gives a really brief overview of the architecture of MIPS R10000, with emphasis on the timing aspects. [3] and [4] describes the MIPS R10000 architecture in more detail.

MIPS R10000 is a superscalar architecture. It has a fetch width of 4 instructions. It stores the instructions it fetches into the instruction buffer which has a capacity of 8. The processor can decode 4 instructions at a time. But decoding stops when it encounters a predicted-taken branch or jump instruction. Also, in the case of a predicted-taken branch or a jump instruction, the subsequent sets of 4 instructions are also discarded. Since the processor speculatively issues instructions to execute, there must be some mechanism to roll back in case of a misprediction. So whenever a branch instruction is encountered, the processor saves its microarchitectural state in a branch stack. It has a total of 4 entries in this stack, so it can speculate upto 4 branches till atleast one of them is resolved. Now the decode stage maps each instruction's destination operand into one of the 64 physical registers. So, at any cycle, utmost 4 physical registers will be allotted at the decode stage. It also allocates one ROB entry for each instruction it decodes, thus allotting utmost 4 entries in any cycle. There are a total of 32 ROB entries. Each decoded instruction (except jump instructions) is dispatched into one of the three queues:

1. Integer queue
2. Address queue
3. Floating point queue

Integer queue:

Each instruction that has to be serviced by an integer ALU enters into the integer queue. All the branch instructions whose condition has to be evaluated also enter into this queue. 4 instructions can enter the integer queue at any clock cycle and utmost 2 instructions will be issued. This is because there are only 2 ALUs. Also the two ALUs are asymmetric. Only one of the ALUs can handle shift instructions and can evaluate conditions for branch instructions. The priority of issuing instructions in the queue is based on static priority – the static position in the queue determines the priority rather than the dynamic position with respect to the head of the queue. The ALUs have a latency of 1 cycle and a repeat rate of 1 cycle. So an instruction can be issued to either ALU at every clock cycle.

Address queue:

This queue handles all the load and store instructions, so it maintains the instructions in the order of their dispatch, which is program order. It takes one cycle to perform the virtual address to physical address translation and another cycle to load or

store the physical address to memory, in case of a cache hit. But the load store unit is pipelined, therefore it has a latency of 2 cycles and a repeat rate of 1 cycle, in case of a cache hit. If the load or the store misses the cache, it takes several cycles to service this request during which time the address queue can not issue instructions into the load/store unit.

Floating Point queue:

This queue handles all the floating point instructions. It takes variable number of cycles to execute most of the floating point instructions. This was not implemented as SMIPS does not support floating point instructions.

An instruction can be issued from the queue whenever its source operands are ready (ie completed evaluation) or when the source operands will be ready in the next cycle. For example, every ALU instruction which depend on the previous instruction can be issued in consecutive cycles. Similarly an ALU instruction which depends on the result of a load instruction can be issued speculatively, assuming the load instruction to result in a cache hit. If the speculation is proved to be false, the next ALU instruction will be aborted. In the case of branch mispredict, the entries in all the three issue queues which are speculatively executed after predicting for that corresponding branch will be marked as aborted. After an instruction finishes executing in the functional units, they will update the corresponding ROB entry as done. Program commit proceeds in the original program order, committing 4 instructions at a time. Another thing to note about R10000 architecture is that after a branch mispredict, the instructions are got from the branch cache instead of from the cache. The instructions enter the branch cache instead of being discarded in the case of a different branch prediction. This reduces one cycle latency in the case of branch mispredict. Another point to note is that MIPS ISA has a delay slot after each branch instruction. The MIPS R10000 architecture also allowed this delay slot in order to conform to the ISA, even though the superscalar out-of-order processor hides the branch delay already.

I initially started out with the goal to implement a cycle accurate timing partition for the MIPS R10000 architecture for the SMIPS ISA using the HAsim infrastructure. But due to lack of sufficient time, I ended up implementing a slightly different architecture. Nevertheless, it is still a 4-wide superscalar out-of-order issue processor. In this subsection, I will describe the differences between the architecture whose timing partition I implemented, and the MIPS R10000. This architecture will henceforth be called as S10000.

B Implementation of timing partition

Figure 11 shows the high level implementation of the timing partition for the S10000 architecture. The red arrows represent the APorts used to communicate between

the modules of the timing partition, whereas the black arrows represent the request and response messages between the timing partition and the functional partition. The dotted lines separate the timing partition from the functional partition. All the modules in the timing partition were implemented by me, whereas the whole of the functional partition was implemented by Michael Pellauer.

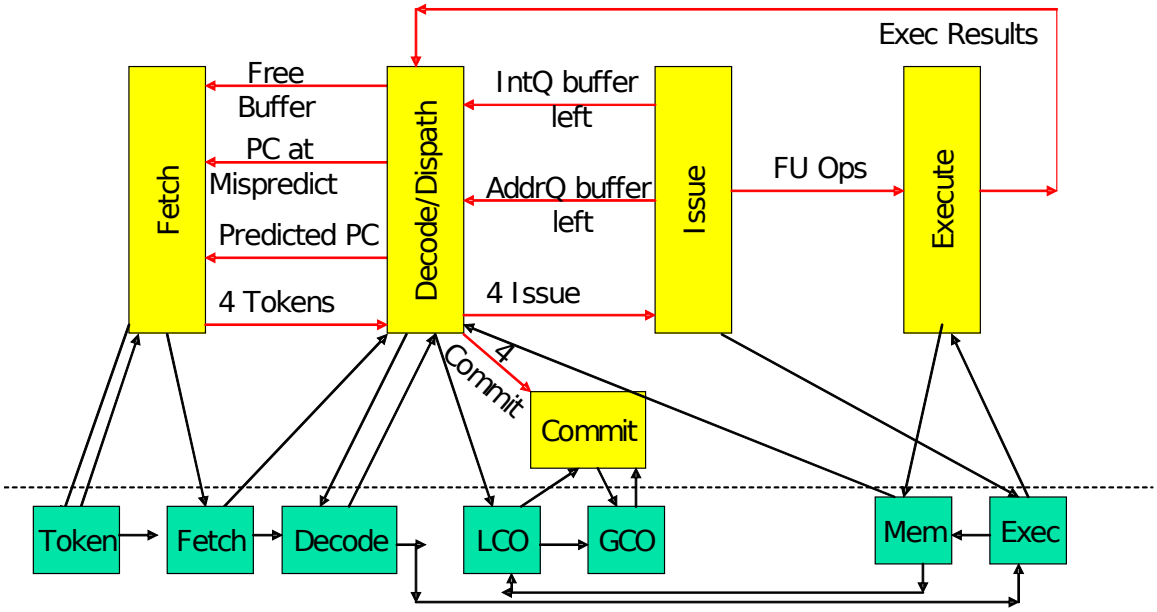

Figure 11
S10000 Simulator – Timing and Functional partitions

Fetch Stage:

The fetch stage in the timing model starts when it receives information on the amount of free buffer space available in the instruction buffer (which has a total of 8 spots), and when it receives information about the branch prediction and whether the instruction was mispredicted, both from the decode/dispatch stage. It sends utmost 4 token requests to the Token unit of the functional partition. It then sends these 4 tokens to the decode/dispatch stage. This unit is pipelined in the sense that the instruction fetch requests will be sent every clock cycle.

Decode/Dispatch Stage:

The actual instruction is received in the decode/dispatch stage so that it can proceed in parallel with the fetch stage. This stage stores the incoming tokens and the corresponding instructions into the instruction buffer. It also sends a decode request to the Decode unit of the functional partition so that it can determine the dependency information of that instruction and dispatch the instructions correctly, in the sense of

filling the ready bits of the source operands depending on the dependency information received from the decode unit of the function partition. The decode/dispatch stage also dispatches the instructions to the issue stage, and fills in an ROB entry, which essentially maps the token to the physical register and also has flags to determine if the instruction is a branch or not, if the branch was predicted to be taken or not, if the instruction is a JR or JALR instruction and the speculated jump address. The decode stage takes as input the number of free slots left in the integer queue and the address queue of the issue stage so that it can make a decision on whether to decode a particular instruction or not. The results of the execute unit, along with the corresponding replies (which are acknowledgements) from the memory unit of the functional partition, goes into the decode/dispatch stage and is used to update the isComplete bit of the ROB. While updating the isComplete bit in the ROB entry, it also checks to see if the instruction is a valid instruction or not. This is because I don't abort the instructions in the issue queues. Rather, during a branch mispredict, I simply update the window of the ROB (and send a rewind token to the functional partition to rewind to the token corresponding to the mispredicted branch) , and when an instruction is completed from the execute stage, I check to see if it lies within the window. If it lies within the window, the ROB entry corresponding to that instruction is set to Completed, else that instruction is discarded and its token killed. In parallel to all these operations, utmost 4 instructions are passed on to the commit stage for commit, and the corresponding commit requests are sent to the functional partition. This unit exploits all the parallelism available in implementing the decode/dispatch stage and also pipelines its execution completely. For example, processing the instructions from the fetch stage and sending instructions to the commit stage are completely synchronous activities. So they are done in parallel. Similarly updating the ROB entry after receiving the completed instructions from the execute and the former two activities can be done in parallel. And by pipelining, I mean, each instruction (out of the 4 instructions in the superscalar processor) gets processed every clock cycle. All this leads to a substantial increase in throughput of the simulator.

Commit Stage:

This stage accepts instructions to commit from the decode/dispatch stage as well as the local commit responses (acks) from the functional partition and sends global commit requests and receives responses (again acks) from the functional partition for the same. This stage is also pipelined.

Issue Stage:

The issue queue splits the instructions it gets from the decode/dispatch stage into the integer queue and the address queue. It also has a one entry jump queue (not for JR and JALR), which is used to handle the jump instructions, as they also have to be sent through the entire pipeline. This does not affect the timing information as their execution is overlapped with other instructions. The fundamental reason for why this is being done this way is that the functional partition allows only one module of the timing partition to communicate with each module in the functional partition. This means that the execute request can only be sent from the execute stage of the timing partition. In the original

MIPS R10000 architecture, the jumps do not enter the issue queues, but I made a special queue to allow them to enter in order to complete their execution in the functional partition. Each of the 4 instructions entering the issue stage is inserted in either the integer queue or the address queue or the branch queue. The issue stage also maintains a scoreboard which determines the position of the destination register in each of the functional units. Once the source operands of the instructions are ready to be issued, the issue queue employs oldest-first priority to issue instructions into the two ALUs and the Load/Store unit. The first APort connecting the issue stage with the execute stage's ALUs has a latency of 1, whereas the second APort connecting the issue stage with the execute stage's Load/Store unit has a latency of 2. The execute requests to the functional partition is also sent for the issued instructions at this stage.

Execute Stage:

This stage simply forwards the message it receives from the issue stage into the decode/dispatch stage. It also receives the replies from the execute unit of the functional partition, and sends in requests to the memory unit of the functional partition to these instructions.

C Differences in implemented processor from MIPS R10000

- SMIPS ISA has no floating point instructions. So I didn't implement the timing model for the floating point unit, or the floating point queue
- SMIPS compiler doesn't account for a delay slot. So in my timing partition, I didn't have any delay slot.
- The priority of the Integer issue queue is different from that in MIPS R10000
- I didn't implement a branch cache. Instead I simply have a counter which counts till 4 (which is the size of the branch cache), and if the branch counter reaches 4, the decoder will not decode any branch instruction. But this has an effect in the number of cycles wasted during branch mispredict. In the case of the MIPS R10000 architecture, during a branch mispredict, the right instructions are taken from the branch cache which stores the discarded instructions, rather than from the fetch unit. But in the case of my timing model, I had to waste the cycle to fetch instructions from the fetch unit always.
- The JR and JALR instructions had to go through the integer queue. This was a requirement because JR and JALR had to wait for the source operand to be ready before the done bit can be set in the ROB. If the register read gave a value different from that speculated, then it is a misprediction and the ROB has to be restored to the JR or JALR instruction which caused the misprediction. So the result of a prediction is seen as soon as an instruction updates the value of the source register for the JR or JALR instruction. But in HAsim framework, the functional partition does not allow the timing model to look at the values of registers, in order to prevent misuse by the timing partition. So the timing partition has to wait till JR or JALR instruction is executed before it can make a

prediction. Let us say we introduced a new queue to handle JR and JALR instructions. Now this introduces another complexity as the number of JR and JALR instructions in flight will be restricted to the queue size. This means that we will introduce an artificial limit on the number of JR and JALR instructions in flight. Of course we can easily bound the maximum number of JR or JALR instruction in flight (which will be 32), but this solution seems to be a hack and a waste of clock cycles. Currently we are investigating techniques to allow multiple timing partition modules to communicate to a single functional partition module via the request-reply interface. This solves this problem because whenever an instruction writes to the register whose value a JR or JALR reads, then we can bypass the path through the issue and execute stages of the timing partition (which increments the clock cycles as represented by APorts), and instead use a separate path to the execute unit of the functional partition to execute the JR and JALR instructions. If this is done, even the other jump instructions can take this path for execution.

D Simulation results

SMIPS v2's ADDUI test case was run successfully at the time of writing this and the simulation took a total of 239 FPGA cycles to run. The total number of model cycles it took was 7. Also the number of lines of code to write the entire timing model was approximately 1300, in comparison to write the whole of SMIPS processor which took approximately 1200 lines of Bluespec code, excluding the memory or the caches. So writing a timing model in the HAsim framework is relatively an easy task.

References:

1) APorts – Performance modeling of synchronous digital circuits with synchronous digital circuits, Michael Pellauer and Joel Emer
2) AWB – The Asim Architect's Workbench, Joel Emer, Carl Beckmann, Michael Pellauer
3) MIPS R10000 Superscalar Microprocessor, Kenneth C. Yeager, IEEE Micro, Vol. 16, Issue 2, April 1996, Pages 28-40
4) MIPS R10000 Uses Decoupled Architecture: High-Performance Core Will Drive MIPS High-End for Years, Linley Gwennap, Vol. 8, No. 14, October 24, 1994