RC24706 (W0812-053) December 10, 2008
Computer Science

IBM Research Report

IBM PowerPC Design in Bluespec

Kattamuri Ekanadham, Jessica Tseng, Pratap Pattnaik
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

IBM PowerPC Design in Bluespec

Kattamuri Ekanadham, Jessica Tseng, Pratap Pattnaik
(eknath, jhtseng, pratap @ us.ibm.com)

IBM Thomas J. Watson Research Center
Yorktown Heights, New York

December 09, 2008

We describe here the structure and principal components of the design of a multi-threaded powerPC processor using Bluespec. The focus is on the generality and flexibility of structure, using the high-level nature of Bluespec language, so that the resulting design facilitates rapid experimentation with incremental changes to the architecture. Hence, the code is highly parameterized so that the design can be easily tailored to vary number of threads, cores, various table sizes, etc. The implementation presented here is of a very primitive processor that has no cache subsystem and is directly connected to a memory. A preliminary design of an address translation mechanism is included. We describe the structuring of some salient components and give some code fragments to illustrate the flavor of coding style. The complete processor is synthesized successfully and is currently being ported onto an FPGA platform.

1. Overview

The overview of the processor structure is depicted in Figure 1. It consists of an instruction unit (Iunit) and an execution unit (Xunit), each of which is a pipeline of several stages as shown. The two units share a single address translation hardware (TLB) and a single interface to memory. A set of queues (one for each thread) of decoded instructions connect the two units. The Iunit enqueues instructions into it and the Xunit dequeues from it as it dispatches.

The Iunit is a sequence of stages and the passage of a single packet through all the stages of the Iunit accomplishes the task of bringing one instruction block into the machine. This comprises of translating a block address, fetching it from memory, breaking it into instructions, determining the set of instructions from the block that belong to the predicted path of execution, decoding the instructions from that set, cracking complex instructions out of them and finally enqueueing them into the decode queue associated with that thread. The pipeline splices this block processing for various threads in a fair manner. It has the provisions to receive signals from Xunit when a thread is to be reset and restarted at a different address and takes appropriate actions to clear all partial processing of a block when a new block must be started.

The Xunit, similarly is a sequence of stages and the passage of a single packet through all the stages of the Xunit accomplishes the task of executing one instruction of a thread. This comprises of reading the necessary register values, performing any address arithmetic needed to compute effective addresses, translating addresses, accessing the memory for load/store instructions, executing any logical/arithmetic operation for the instruction, storing the result of an instruction in the architectural state of the thread and determining the next instruction the thread must execute. A single stage (commit

stage) is where the instruction is committed. It handles instruction exceptions and aborts a thread. It recognizes interrupts and saves and restores the states of computation. It has a decrementer that can be programmed. It has provisions to handle longer execution delays (such as memory accesses) and re-execute instructions when necessary.

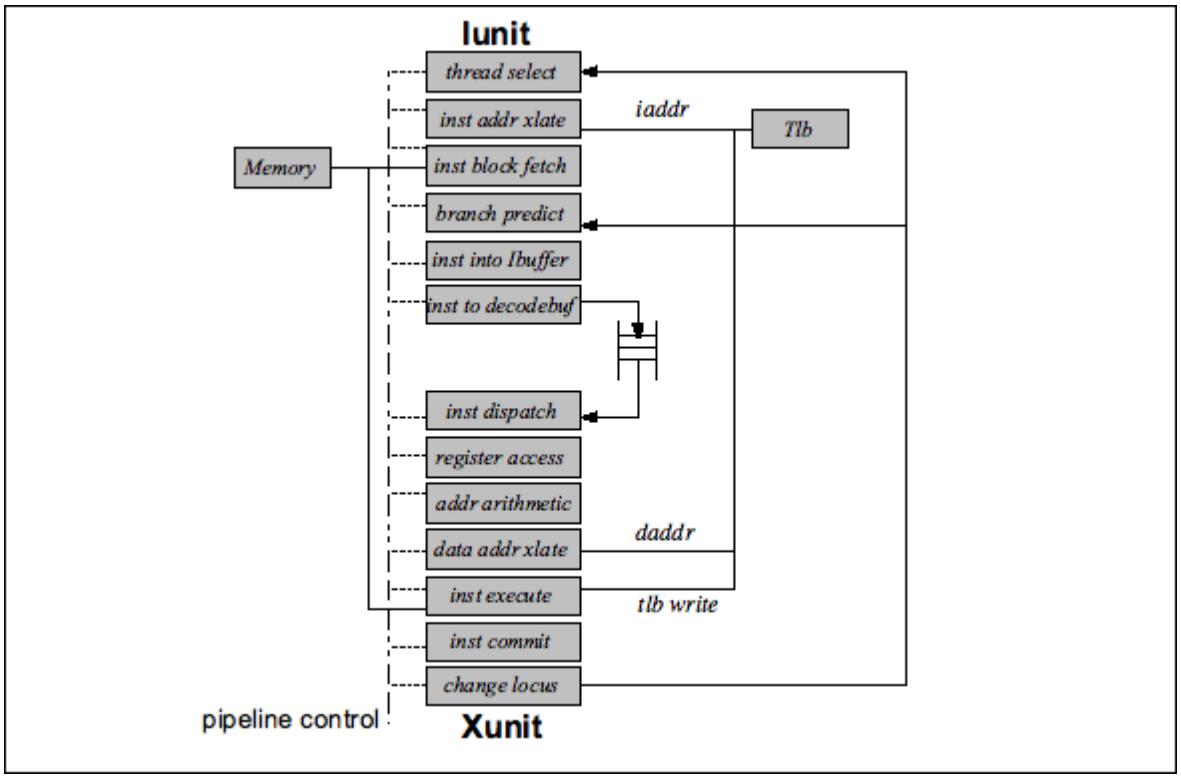

Figure 1: Overview of the Processor Core

2. Pipeline Abstraction

The pipeline is designed as a generic abstraction that controls stages oblivious of the number or nature of the component stages. This facilitates designing each stage independently and stages can be added and removed in a very flexible manner. From the controller's view, the pipeline is a vector of stages through which packets of certain type flow. Each packet represents an instruction carrying all information about its thread, its operation etc. Each stage represents an operation to be performed on the incoming packet and produces an output packet. In addition, the operation performed in a stage may result in altering the status of the threads. To capture this, the pipeline defines the notion of status of a thread as follows:

Thread Status: The status of a thread can take one of 4 possible values: *Active, Suspend, Reset* or *Restart.* Intuitively, an active status indicates that a packet for that thread should be normally processed. A suspend status indicates that processing of any packet for that thread must be abandoned and the thread will have to be reactivated later to re-execute the same instruction. A reset status also cancels all processing for a thread and in addition, it will also discard any partial state maintained for the thread. The thread must then be restarted to start with a new instruction at another address.

Stage Operation: Each stage is viewed as an operator. At the beginning of each cycle, it is provided with an input packet and a status vector indicating the status of each thread as viewed by that stage. At the end of the cycle, the stage returns a result packet and possibly an altered view of the status of each thread, as a result of its operation. The stage is free to maintain and update an internal state during this operation. The controller collects the outputs of all stages, pushes the packets to successive stages and propagates the statuses up the pipeline. The manner in which the status propagation is used to control the thread behavior is described next.

Controlling Threads: In general, several instructions of a thread may be in progress at different stages in the pipeline, in any cycle. The pipeline provides a variety of ways in which the progress of a thread can be controlled. For instance, when a stage *i* returns suspend status for thread *j,* the status is propagated upwards so that all preceding stages discard processing any packets belonging to that thread, while all succeeding stages continue with any packets belonging to that thread. When the suspending stage eventually decides to reactivate the thread, it submits an active status, which is again propagated, so that the top stage resubmits the suspended instruction packet. Similarly, when an execution stage detects a taken branch, it submits a reset status causing all preceding stages to discard all processing for that thread and clean up any partial states. The same mechanism is used to switch a thread upon recognizing an exception or external interrupt.

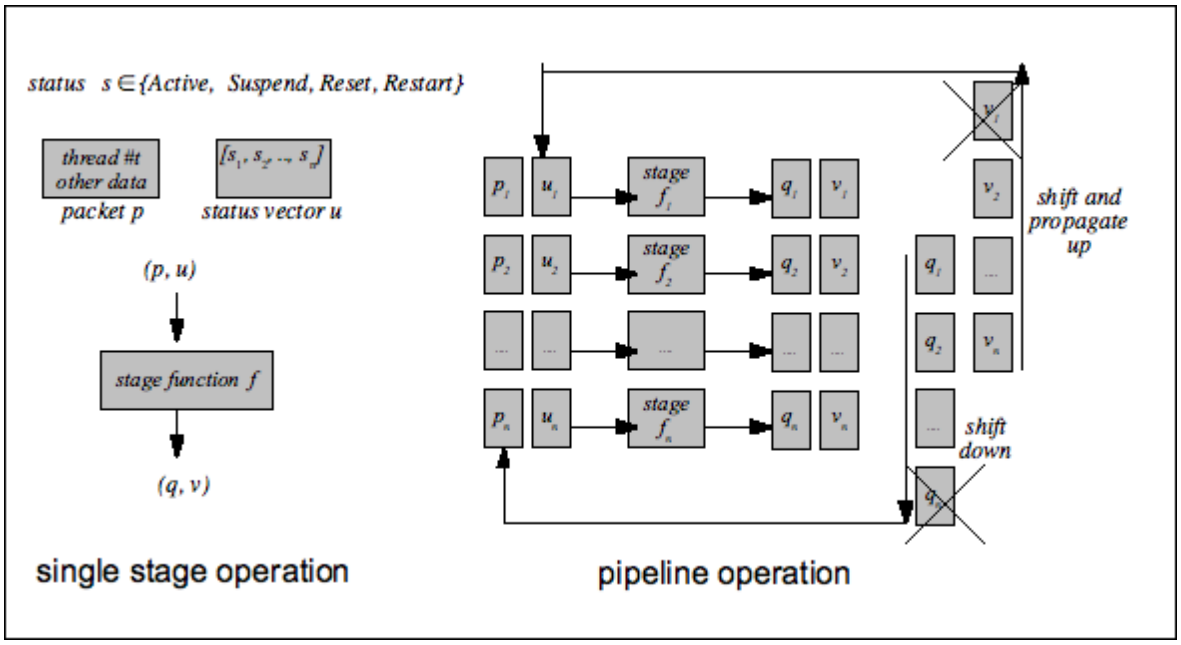

Figure 2: Pipeline Control Schema

The basic operation of the pipeline controller is illustrated in Figure 2. A stage is abstracted by a function, *f,* that takes pair *(p,u)*, of packet *p* and status vector *u* and returns the pair *(q,v)*. The controller operates on a vector of such pairs, one for each stage. It feeds the corresponding pair to each stage and collects the resulting pair. (see code fragment 1) It shifts the packet vector down so that the output packet of a stage is fed to the next stage in the next cycle. For the status vector, it accumulates them upwards, so that stage *i* can see the cumulative effect of statuses set by all stages below *i.* It shifts the resultant vector up and feeds to the corresponding stages in the next cycle. There is provision to inject a new input packet at the top of the pipeline and a new status vector to be injected at the bottom

of the pipeline. This pipeline operation repeats every cycle.

3. Instruction Unit

Iunit Pipeline: The Iunit pipeline may process instruction blocks for different threads concurrently. But the instruction block processing for a given thread is done sequentially does not overlap with fetching of other threads for the same thread. When a thread is selected for instruction fetching, it sends out a suspend status and will not be selected again until that instruction block is fetched, inserted into instruction buffer and its next block address is determined. Throughout this processing, a suspend status gets sent back for that thread, preventing the top stage from selecting that thread again. Since reading instruction memory can take arbitrary amount of time, the following strategy is adapted.

Instruction fetching: The instruction block fetch stage is a finite state machine, and its state is characterized by three boolean variables [present, issued, tobeDiscarded] for each thread. They respectively track whether (a)there is an outstanding block request present for the thread, (b)the request has been issued to memory or not and (c) whether a future response for it must be discarded or not. In each cycle, any incoming packet receives a block request; If no response is being awaited for a previous request, it marks it as present, not issued and notToBe Discarded. Otherwise, it marks as present, not Issued and response to be discarded. Then it selects, in a round-robin fashion, a thread with a request that is present and not issued and for which no response is being awaited. The request is issued and so marked. When a response is received for a thread which is marked to be discarded, it discards the response and the state is so marked. Otherwise, its state is cleared and an output packet is generated with the contents of the block. (see Code Fragment 2). This way, even if a thread is reset while it has an outstanding memory response, it awaits the response, discards it and proceeds with the new request.

Instruction Buffering: Once a block is fetched, the next stage picks one instruction at a time and inserts into Ibuffer. This is done again in a round-robin fashion, so that processing of instructions for various threads is interleaved in a fair manner avoiding starvation. Once an entire block is completed for a thread, its status is set to Active and a new block address is supplied for it. (see Code Fragment 3).

Instruction Decoding: The same sharing strategy is adapted for decoding instructions from Ibuffer and inserting them into decode buffer. The decode buffer is shared with the dispatch stage in the Xunit and the selection criterion for decoder will check for a thread's Ibuffer to be non-empty and its Decode buffer to be non-full. (see Code Fragment 4). In addition, load-multiple and store-multiple instructions are cracked into individual load/store instructions in this stage.

4. Execution Unit

Dispatcher: The dispatcher performs two main functions: it ensures fair scheduling of instructions from all threads in a round-robin fashion and it remembers a short history of instructions issued for

each thread, so that when a thread is resumed after suspension, instructions can be reissued in an orderly fashion. To aid these functions it maintains the structures shown in Figure 3. *GA* is a global array. There is a history array, *HAi,* one for each thread. Each entry in *HAi* can hold an instruction and each entry of *GA* can hold an index into any history array. All arrays are of the same size as the number of entries in the pipeline. The pointers *g, ai, ri* are used as described below. In the normal operation, when an instruction is dispatched for thread *i,* the instruction is stored in *HAi[ai],* the index ai is stored in *GA[g]* and the pointers *g, ai* and r*i* are incremented (circularly). If no instruction is dispatched, then only *g* is advanced. When thread *j* gets suspended by stage *k*, then *rj* is set to the current value in *GA[g-k]*. Next time when thread *j* is selected for dispatch, the instruction at *HAj[rj]* is dispatched and *rj* is advanced. This process continues until *rj* reaches *aj,* after which normal operation is resumed for that thread. (see Code Fragment 5).

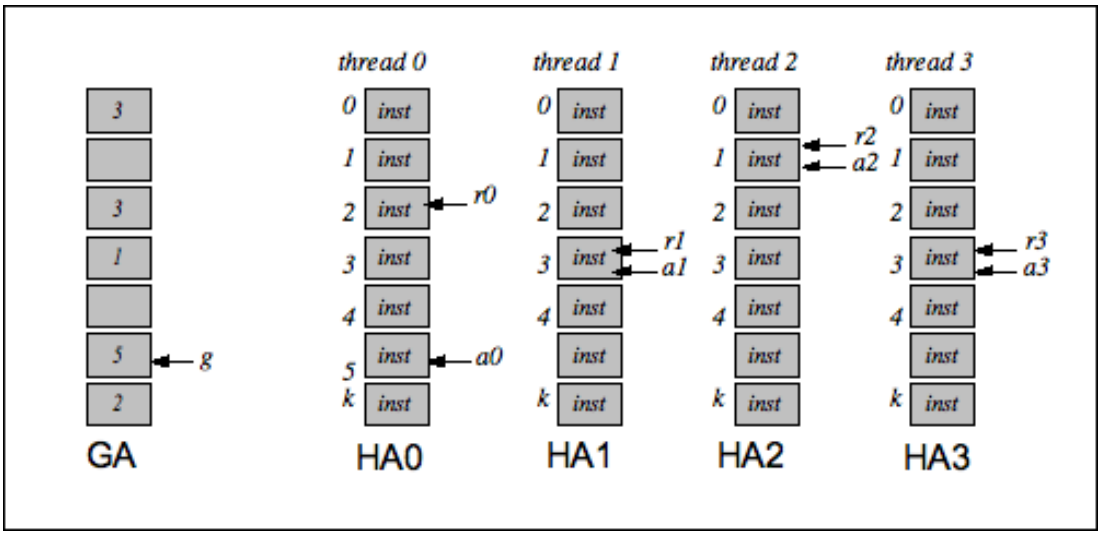

Figure 3: Instruction History maintained by Dispatcher

Register Bypass: The register values are read in the second stage in Xunit and they are used the following two stages, before new values are written into them in the commit stage. This leaves a window during which stale values could be used when stores and loads take place back to back in the pipeline. To overcome this, an array of register values written in the last *k* cycles are stored in an array. In the execute stage, new values are taken from this array, if they can be directly used. Otherwise, the instruction is suspended so that it is resubmitted again.

Memory Accesses: Execution of load/store instructions is designed to tolerate arbitrary latency for memory access. The load/store unit remembers the latest memory request made by each thread and submits them to memory in a round-robin fashion. Once a store request is accepted from a thread, no further memory requests are accepted from that thread until the store is completed. Further requests are simply suspended and retried until they are accepted. When a load request is accepted, the load instruction is suspended and retried, while the load is in progress. Data received from memory is buffered in the load-store unit and is delivered when the load is retried next. Store requests will erase any pending data in the buffers to preserve consistency. Partial word writes are implemented by first reading the memory and writing back the updated words. No memory request is permitted to intervene between such read and write involved in a partial store.

Address Translation: A central table of entries is maintained to translate both instruction and data memory addresses. Multiple page sizes are supported. All the entries in the translation table are searched in parallel to find a match. Address translation faults generate an interrupt to resolve the fault. Multiple requests to the translation table are ordered by priority and are handled one at a time. Writes to the translation table take highest priority. Then data address translations are taken up. Instruction address translations have least priority.

Decrementer: A programmable decrementer is provided that decrements a count in each cycle and raises an interrupt when it reaches zero.

5. Validation Strategy

The Bluespec compiler produces equivalent versions of implementations in C and Verilog. The Bluesim simulator can be used to simulate the C version and observe the inputs and outputs. This path is used to debug the core written in Bluespec. The memory is implemented as a file that is accessed by the Bluesim simulator. PowerPC Binary file is supplied as initial contents of the memory. A rudimentary interface is built in the core to read the contents of a designated area in the memory to initialize its registers before the core starts to run and also to dump the contents of the registers in that area when the execution is stopped. Using this one can specify a complete initial state of the system and a program and obtain the complete final state after the program completes its execution of a specified number of instructions.

IBM has developed a sophisticated processor simulation facility (called Mambo) and PowerPC processor designs are available that run on Mambo. Given a program and an initial state of the system, Mambo has the capability to run behavior simulation and provide the final state of the system after running a specified number of instructions. Using this facility, we compared the final states from Mambo and Bluesim simulators for a given initial state. While this is, by no means, an exhaustive unit test, this provided us a preliminary validation for various excerpts from Linux code and other micro benchmarks.

6. FPGA Implementation

For the initial prototyping, we use a Xilinx Virtex-5 LX330 FPGA device, with an 8M SRAM on board (known as Stinger-3), as shown in Figure 4. The FPGA board is controlled by a host computer (a laptop PC). The host computer is connected to the board via 2 ports. The USB port connection is used as a command interface to control the core and the RS232 port is used for the UART connection for kernel display and input, when the core is running.

The FPGA board is populated with a processor core, an 8MB SRAM and 4 interfaces (cmd, mem, usb, uart) as shown in Figure 4. The command interface accepts commands from the host and returns responses. This can be used to operate the core in two modes: probe mode and run mode. In the probe mode, the core state (registers and memory) can be examined and set to specified values via the

command interface. In the run mode the core is run for a specified number of instructions. The three interfaces, usb, uart and mem, respectively interface with the 3 external devices and transfer signals in both directions, crossing the clock boundaries as needed. The memory interface provides a quad-word half-duplex data path to the SRAM, together with controls to read/write individual words on the quad-word path. Currently the clocks are adjusted to provide a fully pipelined interface to the memory with a 3 cycle latency for any access.

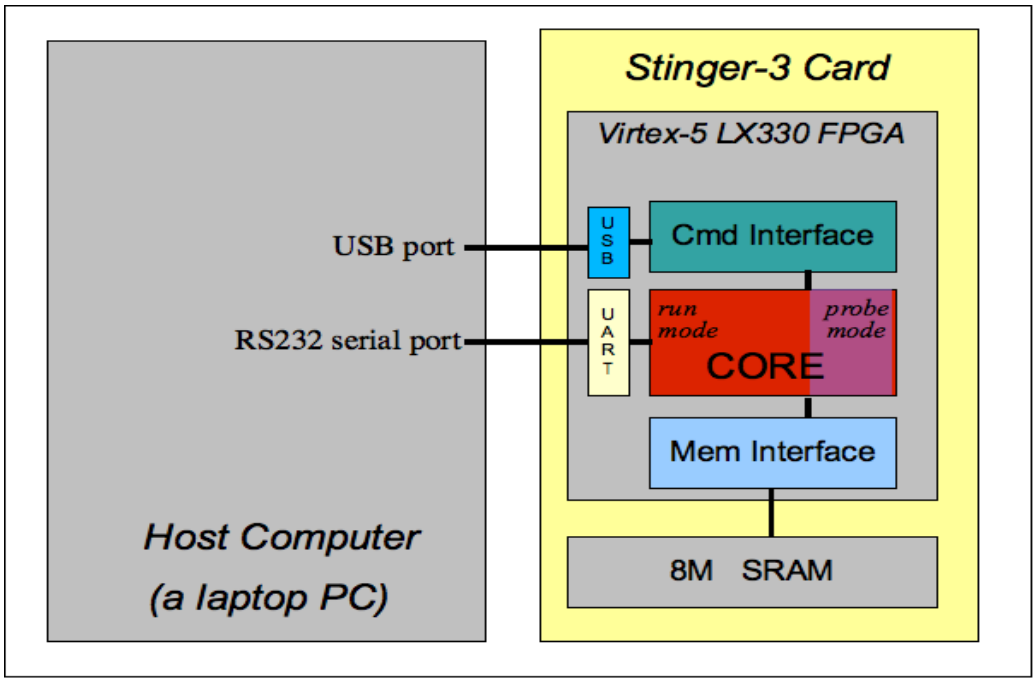

Figure 4: FPGA board and its contents

The preliminary sizes of the Bluespec programs, the Verilog programs generated for them and the area occupied by their synthesized versions are shown in Table 1. The table also shows how the FPGA resource usage vary as the number of threads per core is increased without any optimization. Xilinx Virtex-5 LX330 FPGA device includes 207,360 internal fabric flip-flops and 207,360 6-input look-up tables (LUT). For the single thread configuration, 11% of available flip-flops and 32% of available LUTs are consumed; these numbers increase 63% and 56% respectively to 18% and 50% for the four-thread configuration.

Component	Lines in Bluespec	Lines in Verilog	Flip-flops in FPGA	LUTs in FPGA
Single Thread				
Core + Cmd Interface	12,433	91,312	22,360	62,915
4 Threads				
Core + Cmd Interface	12,433	166,171	36,469	98,443
Interfaces Written in VHDL				
Memory Interface	vhdl	296	302	153
Usb Interface	vhdl	311	145	92
Uart Interface	vhdl	5,451	710	887
Interconnect all components in the FPGA	vhdl	439	33	3,249

Table 1: Sizes of Programs and Corresponding Area occupied in FPGA

7. Appendix: Illustrative Code Fragments

We provide Bluespec code fragments that correspond to certain descriptions given in the text. These are excerpts from actual code and are illustrative of the coding style.

```
// code Fragment 1
// run one step of the pipeline
function ActionValue#(stagePkt)
  applyStageFunction(StageFunction#(stagePkt) f, stagePkt x); return f(x);
endfunction
function ActionValue#(StagePktVec#(nStages, stagePkt))
  runPipe(StageFunctionVec#(nStages, stagePkt) fvec, StagePktVec#(nStages, stagePkt) pvec);
  return zipWithM(applyStageFunction, fvec, pvec);
endfunction


//   given initialStatus and an initial status vector[0..nStages] this function
//   propagates the max status upwards for each stage and also captures the largest stage
//   numer that set the max status. It returns that stage number and the propagated status vector
function Tuple2#(TypeStageNum#(nStages), Vector#(nStages, TypeStatus))
  propagateStatusUpward(TypeStatus initStatus, Vector#(nStages, TypeStatus) svec);
      Tuple2#(TypeStatus, TypeStageNum#(nStages)) initPair = tuple2(initStatus, fromInteger(valueof(nStages)-1));
```

```
    Vector#(nStages, TypeStageNum#(nStages)) snumvec = genWith(fromInteger);
    Vector#(nStages, Tuple2#(TypeStatus, TypeStageNum#(nStages))) pairVec = zip(svec, snumvec);
    Vector#(nStages, Tuple2#(TypeStatus, TypeStageNum#(nStages))) proppedPairVec =
                                                    sscanr(maxPairStatusNum, initPair, pairVec);
    match {.newsvec, .numvec} = unzip(proppedPairVec);
        return tuple2(numvec[0], newsvec);
endfunction
```

// code Fragment 2

```
//  select and issue a new inst fetch request to memory
// if not busy, select next thread and submit another request
    if  (icacheRequestQueue.notFull()) begin
          TypeThreadNum w = advanceThreadNum(lastThreadServed);
          TypeMemReq r = ?;
          Maybe#(TypeThreadNum) maybeS = tagged Invalid;
          TypeBoolVec eligible = zipWith3(firstTrueRestFalse, newReqValidVec, newReqIssuedVec, newReqDiscardVec);
          for(Integer i=0;i<valueof(NTHREADS);i=i+1) begin
            if (maybeS matches tagged Invalid &&& eligible[w]) begin
              maybeS= tagged Valid w;
              r = newReqVec[w];
            end
            w = advanceThreadNum(w);
          end
          if (maybeS matches tagged Valid .s) begin
            r.addr = r.addr & (~('hf)); //send the addr of quad-word containing the desired word
            TypeTaggedMemReq tr =TypeTaggedMemReq{tag: TypeMemTag{isFromIU:1, isStore:0, tnum:s}, memReq:r};
            newLastThreadServed = s;
            newReqIssuedVec[s] = True;
            icacheRequestQueue.enq(tr);
          end
    end


   // If Icache resp arrived, note the results
    if  (respWire.wget matches tagged Valid .resp) begin
      TypeMemTag tag = resp.tag;
      TypeThreadNum t = tag.tnum;
          if (newReqDiscardVec[t]) begin
            newReqDiscardVec[t] = False;
          end
          else begin
            retPkt.validPkt = True;
            retPkt.tnum = t;
            retPkt.addr =  newReqVec[t].addr;
            retPkt.tlbMiss = (resp.memResp.dataValid != 1);
            retPkt.ivec = resp.memResp.data;
            // mark selected instructions ... code deleted  for brevity
            newReqValidVec[t] = False; // this req is done
            newReqIssuedVec[t] = False;
            newReqDiscardVec[t] = False;
          end
    end
```

// code Fragment 3

```
//  select an eligible thread that has an inst to enter into its IBuffer

    function Bool eligibleForXfer(TypePendingIUstagePkts pendingIUpkts,TypeThreadNum t);
        return (isIUpktValid(pendingIUpkts[t]) && ibuf[t].notFull());
    endfunction

    TypeMaybeThreadNum maybeT = selectThread(map(eligibleForXfer(pendingIUpkts), threadNums),
iBufferNextTnum);
    maybeTtoStoreIbuf <= maybeT;

    for (Integer k=0; k<valueof(NTHREADS); k=k+1)
        if (maybeT matches tagged Valid .tt &&& tt==fromInteger(k)) begin
          TypeThreadNum t = fromInteger(k);
          TypeIUstagePkt q = pendingIUpkts[t];

          TypeIbufEntry e = TypeIbufEntry{addr: q.addr,inst: q.ivec[0],isBranch: q.isBranch[0],
                                          brTaken: q.isTaken[0],tlbMiss: q.tlbMiss};
          ibufEntryToStoreIbuf <= e;
          iBufferNextTnum <= advanceThreadNum(t);
          q.addr = q.addr + 4;
          q.ivec = shiftInAtN(q.ivec, unpack(0));
          q.isBranch = shiftInAtN(q.isBranch, unpack(0));
          q.isTaken = shiftInAtN(q.isTaken, unpack(0));
          q.isSelected = shiftInAtN(q.isSelected, unpack(0));
          Bool noMore = (q.tlbMiss || !q.isSelected[0]);

          if (noMore) begin
            q.validPkt = False;
            retPkt.svec[t] = Active;
            newIarVec[t] = tagged Valid q.addr;
          end
          pendingIUpkts[t] = q;
        end
```

// code Fragment 4

```
// decodes an instruction and inserts into decode buffer;
// LoadMultiple and StoreMultiple instructions are cracked into individual loads and stores
  method ActionValue#(TypeIUstagePkt) ibufToDibuf(TypeIUstagePkt  p);
    function Bool isEligible(TypeStatus s, TypeThreadNum t);
        return ((s==Active) && ibuf[t].notEmpty() && dibuf[t].notFull()); endfunction
    if (crackingThread matches tagged Valid .ct &&& dibuf[ct].notFull()) begin
        wireThread <= crackingThread;
        wireThreadReset <= (p.svec[ct] != Active);
    end
    else begin
      TypeBoolVec eligible = zipWith(isEligible, p.svec, threadNums);
      TypeMaybeThreadNum maybeT = selectThread(eligible, iDecodeNextTnum);
```

```
        maybeTtoXferIbufToDibuf <= maybeT;
    end
    TypeIUstagePkt retPkt = ?;
    retPkt.validPkt = False;
    retPkt.svec = replicate(Active);
    retPkt.avec = replicate(tagged Invalid);
    return retPkt;
  endmethod: ibufToDibuf
```

```
// code Fragment 5

    // For each thread that is reactivated after suspension, roll back its log (based on the
    // stage that has suspended it) and set resuming index to the proper place in its log
    for(Integer i=0;i<valueof(NTHREADS);i=i+1) begin
        TypeThreadNum it = fromInteger(i);
        if (  ( (threadDispatchStatusVec[it]==Active) || (threadDispatchStatusVec[it]==Restart) )
            && (inpStatusVec[it]==Suspend)) begin
          TypeXUstageNum y = moveBackXUstageNumBy(circularIndex, inpStageNums[it]);
          TypeXUstageNum z = circularIndexArray[y];
          newResumingIndex[it] = z;
          suspended[it] = tagged Valid inpStageNums[it];
          //$display("*** suspend[%d] [c=%d, stg=%d, issued at c=%d]resume at %d[@0x%h] ***",
          //          it,circularIndex, inpStageNums[it], y, z,dispatchedInstLog[it][z].addr);
        end
    end


    Maybe#(TypeThreadNum) maybeT = selectThread(eligible, nextDispatchThread);

    // prepare a return packet (this stage never alters the state of any thread)
    TypeXUstagePkt retPkt = ?;
    retPkt.svec = replicate(Active);
    retPkt.avec = replicate(tagged Invalid);
    retPkt.validPkt = isValid(maybeT);
    TypeMaybeThreadNum issued = tagged Invalid;
    TypeMaybeThreadNum resumed = tagged Invalid;
    if (isValid(maybeT)) begin
        TypeThreadNum t = fromMaybe(0, maybeT);
        nextDispatchThread <= advanceThreadNum(t);
        TypeXUstageNum x = ?;
        TypeDecodedIbufEntry di = ?;
        //two cases: is the inst being taken from dibuf or am I re-executing a suspended inst?
        if (revisedStatusVec[t] == Active) begin
          x = activeIndex[t];
          di = dibuf[t].first();
          dibuf[t].deq();
          newLog[t][x] = di;
          newActiveIndex[t] = advanceXUstageNum(x);
          issued = tagged Valid t; //$display("*** issue[%d] %d[@0x%h] ***",t, x, di.addr);
        end
```

```
            else begin
              x = newResumingIndex[t];
              di = dispatchedInstLog[t][x];
              newResumingIndex[t] = advanceXUstageNum(x);
              if (newResumingIndex[t] == activeIndex[t]) revisedStatusVec[t] = Active;
              resumed = tagged Valid t; //$display("*** reissue[%d] %d[@0x%h] ***",t, x, di.addr);
            end
            circularIndexArray[circularIndex] <= x;
    end
```

```
// code Fragment 6

// bank of register values written during the recent past
  Reg#(Vector#(BypassLevels, Vector#(WritePorts, TypeMaybeRegData))) bypassVec  <- mkReg(replicate(replicate(tagged
Invalid)));


  function Maybe#(DataD) selectValid(Maybe#(DataD) a, Maybe#(DataD) b);
    return (isValid(a) ? a : b);
  endfunction

  function Maybe#(DataD) hasNewValue(TypeThreadNum t, RegName r, TypeMaybeRegData mbd);
    return (   mbd matches tagged Valid .s &&&
               s matches tagged TypeRegData {t:.ut, r:.ur, val:.uval} &&&
                        (ur == r) &&&
               (ut==t)
             ? tagged Valid uval : tagged Invalid);
  endfunction

  function Maybe#(DataD) hasNewValueAtAnyPort(TypeThreadNum t, RegName r, Vector#(WritePorts,
TypeMaybeRegData) mbdVec);
    Vector#(WritePorts, Maybe#(DataD)) newValVec = map(hasNewValue(t,r), mbdVec);
    return foldl(selectValid, tagged Invalid, newValVec);
  endfunction

  function Maybe#(DataD) regHasNewValueAtAnyLevel(TypeThreadNum t, RegName r);
    Vector#(BypassLevels, Maybe#(DataD)) newValVec = map(hasNewValueAtAnyPort(t,r), bypassVec);
    return foldl(selectValid, tagged Invalid, newValVec);
  endfunction


  function Vector#(BypassLevels, Vector#(WritePorts, TypeMaybeRegData)) recordRegStores(TypeXUstagePkt p);
    Vector#(WritePorts, TypeMaybeRegData) newRegData = replicate(tagged Invalid);
    if (p.validPkt && p.inst.rdWriteValid) newRegData[0] = tagged Valid TypeRegData {t:p.tnum, r:p.inst.rd,
val:p.rdValueNew};
    if (p.validPkt && p.inst.rs1WriteValid) newRegData[1] = tagged Valid TypeRegData {t:p.tnum, r:p.inst.rs1,
val:p.rs1ValueNew};
    Vector#(1, Vector#(WritePorts, TypeMaybeRegData)) singleton; singleton[0] = newRegData;
    return append(singleton, init(bypassVec));
  endfunction
```