Organization and implementation of the register-renaming mapper for out-of-order IBM POWER4 processors

T. N. Buti
R. G. McDonald
Z. Khwaja
A. Ambekar
H. Q. Le
W. E. Burky
B. Williams

We present a new nonconventional approach for designing and organizing register rename mappers that can be applied in modern out-of-order processor chips. A content-addressable memory (CAM) configuration optimal for such a register mapper application was developed. The structure of the CAM and search engine, described in this paper, facilitates the implementation of the register mapper as a group of custom arrays. Each array is dedicated to executing a specific function. Among the functions we implemented are allocation of registers, maintaining the register map and status, source lookup, saving a shadow copy of the register map, and freeing up of registers. We made a novel implementation of the register mapper to provide rename resources for the IBM POWER4™ chip, which provides the processing power for the IBM eServer™ p690. Such register renaming allows for a high level of concurrency in the pipeline and contributes to superior machine performance.

Introduction

To increase the performance leverage of present-day superscalar pipelined microprocessors beyond technology scaling, one needs to maximize the concurrency and overlap in instruction processing. Microarchitectural techniques for instruction-level parallelism can be used to achieve increased concurrency in instruction processing [1–3]. *Out-of-order execution* and *speculative execution* are two powerful techniques that are exploited in modern high-performance processors to increase the amount of concurrency [4–7]. If the operand data is ready and the required execution resources are free, more concurrency in the pipeline and more performance can be achieved by allowing instructions to be executed out of order. However, while the instructions are processed out of order, they are forced to be committed in program order, which preserves the succession in the architectural states of the machine.

In speculative execution, predictions are made about instructions after branches and are allowed to be speculatively processed in parallel with other instructions. This also increases concurrency and improves performance. If the prediction was false, the speculatively executed instructions are flushed and not committed.

However, to apply these microarchitectural techniques, one has to overcome the instruction data-dependence constraints. These artificial dependences are created by reuse of limited architectural register and memory storage. Such false dependences include *write after read* (WAR) and *write after write* (WAW). A WAR occurs when an instruction that writes a new value must wait for all preceding instructions to read the old value. A WAW happens when more than one instruction is written to the same register or memory location. Executing such instructions out of order overwrites the value of the register produced by one instruction before it might have been read by a subsequent one. Therefore, these false data dependences must be eliminated before one can make use of out-of-order and speculative executions.

These dependences and the associated ordering constraints would not occur if a different register name were assigned every time an instruction writes a new value. By applying register renaming operations, each destination architectural (logical) register name is

0018-8646/05/$5.00 © 2005 IBM

mapped into a unique physical register location in the register file. This, in turn, eliminates all of the false dependences [4]. When an instruction is decoded, its destination logical register number is mapped into a physical register location that is not currently assigned to a logical register. The destination logical register is said to be renamed to the designated physical register. The assigned physical register is therefore removed from the list of free physical registers. All subsequent references to that destination register will point to the same physical register until another instruction that writes to the same logical register is decoded. At that time, the logical register is renamed to a different physical location selected from the free list, and the map is updated to enter the new logical-to-physical mapping.

The physical register of old mappings is returned to the free list to be reused once their values are no longer needed. At the same time, the renaming also provides a mapping table to look up the physical registers assigned to the source logical registers of the instruction. The source operand values are read from these physical locations in the register file. If the free list does not have enough registers, the instruction dispatch is suspended until the needed registers become available. A shadow copy of the register state can also be kept in the register mapper. When an instruction flush occurs, the shadow map is used to restore the register state prior to the flush point so that the machine can resume execution. Thus, it is clear that to facilitate the application of out-of-order and speculative executions to gain machine performance, a register renaming function must be implemented.

In the next section, we describe the register mapper high-level algorithm implemented in the IBM POWER4* machine. The functions, register states and transitions, and logical facilities of the mapper are briefly outlined. Following that, we give a detailed account of the register mapping configuration we applied, we describe our new register mapper organization, we describe the circuit and logic implementation, we discuss instruction dispatch, and, in the final section, we briefly summarize the rename resources used in the POWER4 chip (IBM eServer* p690).

Register mapper high-level algorithm

Functions

Register mappers are implemented in high-performance out-of-order machines to manage a large set of physical registers within an associated register file. In the present POWER4 processor, a dedicated mapper is custom-designed for each type of renamable register file. Renaming was implemented for files for general-purpose registers (GPRs), floating-point registers (FPRs), exception registers (XERs), condition registers (CRs), and control/link registers (CTR/LRs). The register mapper presented here is designed to perform many functions, including the following.

- *Allocation of registers:* It allocates new registers during instruction dispatch for each instruction that writes a new result to a target register. Registers are allocated from a physical pool of registers contained in a particular register file. At any given time, a portion of the registers are being used to hold committed register values, and the rest may be used to hold speculative register results.
- *Register renaming and maintaining a register map:* The mapper performs register renaming, which allows multiple writes to the same logical register. A register map is maintained to associate physical registers with logical registers. A custom content-addressable memory (CAM) is designed to hold the register map. Registers allocated for the dispatched instructions are used to update the map.
- *Source register lookup:* These CAM maps must be searched during instruction dispatch to locate the physical registers that hold the latest results for the source logical register of each instruction.
- *Saving of shadow register maps:* Shadow register maps are also maintained to hold previous mappings and replacement information for each dispatched instruction group until it is completed or flushed. When groups of instructions complete, the results that they produce are committed, and registers that hold older results for the same logical registers are released. Register results do not have to be moved when they are committed. If a group flush occurs, all speculatively assigned registers for the instructions being canceled must be released and made available for reuse. A previous register map must also be restored.
- *Selection of free registers:* The mapper generates and maintains a list of free registers. It preallocates them into an allocation buffer in order to provide early information for future dispatches.
- *Maintaining of register status:* In addition to register maps, some amount of register status information must also be kept in the register mapper. A "ready" or "w" bit indicates whether or not register data is available (in the register file or off a bypass). The instruction issue queue logic indicates when a register is ready. A register may be tentatively ready, depending upon whether a load hits or misses. A load miss could require the status of a register to be reset. Special shifting "dl0 and dl1" bits are also kept and used to confirm that valid load data is delivered at the appropriate time. The instruction issue queue logic provides this register status information (w, dl0, dl1) to the register mapper when a result is about to be written into the register file.

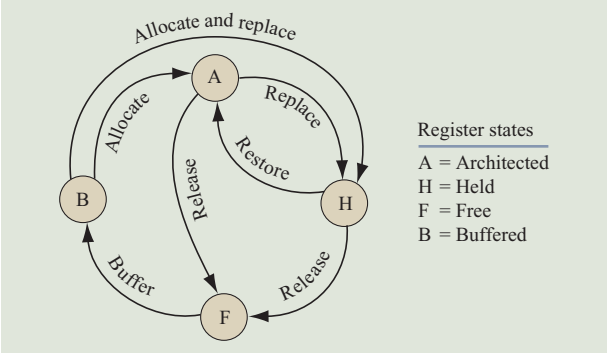

Figure 1

Register logical states and transitions.

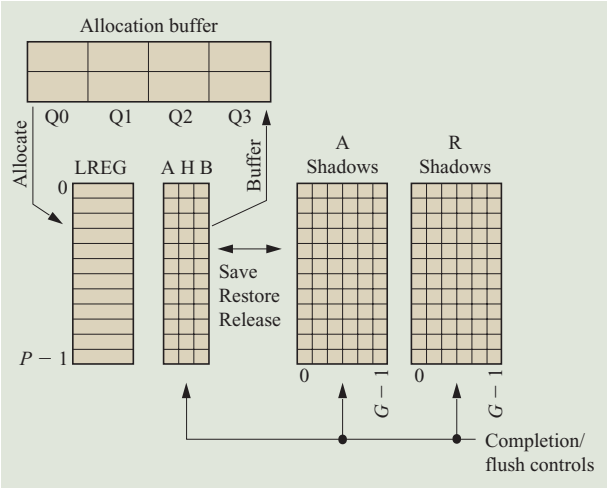

Figure 2

Logical facilities for high-level mapper algorithm. (*P*: number of physical registers; *G*: number of instruction groups; Q: mapper quadrant.)

Register states and transitions

Each physical register may be in one of four logical states, and transitions through these four different states, as depicted in **Figure 1**. A subset of registers that correspond to the initial logical values begin in the architected state. The rest of the registers begin in the free state. A constant number of registers are always in the architected state (one for each logical register). The others may be in any of the remaining states. As instructions are dispatched, new registers are allocated. Registers are replaced when a subsequent instruction writes to the same logical register. Some registers may be replaced by subsequent instructions within the same group and are allocated and replaced at the same time. When replaced, registers transition to the held state. When the replacement becomes committed, the registers are released.

Two events—completion and flush—cause registers to be released or restored. When a group of instructions completes, the registers that it replaced are released. When a group is flushed, the registers that it allocated are released. In addition, when a flush occurs, the registers that were in the architected state just prior to the first flushed group are restored to the architected state. In the current design, the three states—architected (A), held (H), and buffered (B)—are explicitly kept in the mapper arrays. The free state is implicit when none of the other state bits are set. Registers that are in the free state may be selected and moved to an allocation buffer. When this occurs, the registers transition to the buffered state. These buffered registers are then available to be allocated for new instructions.

Logical facilities

The logical facilities used for the high-level algorithm of the mapper are illustrated in **Figure 2**. The actual implementation requires additional state and facilities, discussed in detail later in this paper. In Figure 2, *P* represents the number of physical registers in the register set within an associated register file managed by the mapper. *G* represents the number of instruction groups that may be active in the completion buffer. Q refers to the mapper quadrant. The LREG array records the logical register identifier associated with each physical register in the machine. This information is used to locate source registers and replaced registers during dispatch. A vector of A, H, and B bits indicate the current state of each register. The shadow arrays (A Shadows and R Shadows) keep information about old architected state maps and replacement vectors for each instruction group that has been dispatched but not yet completed. If a flush occurs, this information is needed to restore the proper state. When completion occurs, this information is used to release the appropriate registers. The allocation buffer holds registers that are ready to be allocated.

Register map configuration

In high-performance out-of-order machines, many instructions are dispatched each cycle. In the present machine, four instructions plus one branch are dispatched each clock cycle. This requires simultaneous execution of a fairly large (>12) number of searches of logical source and destination registers in the register map. In the POWER4 processor, within a single clock cycle, up to 16 logical registers must be looked up in the content of the mapper register map array [16 = 4 instructions × (3 sources + 1 destination)]. CAM circuits are usually used to implement the register map. A CAM configuration optimized for such a register mapper

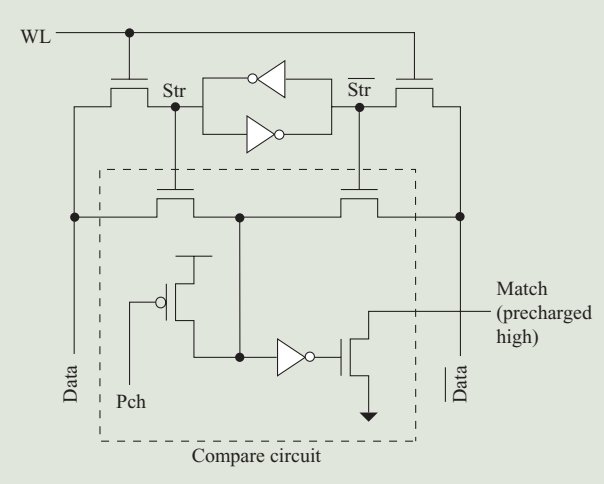

Figure 3

Basic single-compare CAM cell. (Data: incoming bits of data; Str: cell content; WL: word-line signal; Pch: precharge signal; Match: bit-wise compares.)

application requires a nonconventional approach to designing an area-efficient CAM circuit topology and for organizing the register mapper. This challenging task is discussed next in detail.

Conventional approaches

Single-compare CAM cell

A basic single-compare CAM cell [8, 9] is shown in **Figure 3**. It is a simple structure that can store, read, and write 1-bit data. It can also compare a single incoming bit of data (*Data*) against its content (*Str*) and indicate whether or not it matches its content. The CAM array consists of a fixed number of *word rows*, and each word row (CAM word) has the same number of CAM bits (one CAM entry). The CAM array is supported by word-row and bit-column logic to update and access the CAM content. The match operation generates a match line if all of the bits in the search pattern match all of the bits in one CAM entry. The bit-wise compares (matches) in one CAM entry are gated together by AND to produce a match. The output match line is usually used to enable encoding and other readout circuits. Note that only a single CAM search can be performed at a time with this circuit topology. In the register mapping application, the number of word rows is set equal to the number of physical registers available in the register pool. The pattern of CAM bits in a word row is the binary representation of the logical registers used in the instruction sets. The mapping implemented in the CAM array defines the associations of the logical registers with the actual physical registers. This association can also be dynamically updated during instruction dispatch. The output match line is encoded to broadcast the matched physical register (and in our present application, also used to enable the readout of the status bits of a special register). Such additional circuits are placed outside and nearby the CAM array.

Multicompare CAM cell

The CAM cell in Figure 3 is capable of performing one search (compare) at a time. For a high-performance processor, numerous (>12) different searches (compares) must be performed simultaneously against each CAM entry in a single clock cycle. A CAM structure with multicompare CAM cells is required to accommodate such a large number of searches. In this case, multiple match lines are needed for each CAM entry: one match line for each search per CAM entry. All of these match lines must be driven by the same CAM entry. To obtain this, one must integrate into the CAM cell structure as many bit-wise compare circuits as the CAM searches to be conducted in one clock cycle. Theoretically, this can be accomplished by simply integrating the required number of compare circuits (similar to the one in Figure 3) into the CAM cell topology shown in the figure.

Each compare has its own data and $\overline{\text{data}}$ lines, but all compare circuits are connected to the same cell storage nodes (str and $\overline{\text{str}}$ in Figure 3). However, the overhead of running this many tens of bit-wise compares and match lines across each CAM entry would make the CAM cell and CAM entry area far too large to be used in practical chip design. In such an approach, the required compare circuits can be integrated to form a vertical or horizontal stack. If 12 or more of the bit-wise compare circuits are added to form a vertical stack, this would increase the height of the CAM cell by more than one order of magnitude compared with the single-compare CAM cell of Figure 3. It would be impossible to accommodate such a CAM size in a practical chip design. On the other hand, the required compare circuits can be integrated into the CAM cell to form a horizontal stack of bit-wise compare lines. In this case many tens of compare lines must run across the CAM entry to produce the match lines corresponding to the search vectors presented to the CAM array. The number of compare lines is equal to the number of CAM bits per entry times the total number of search vectors (=72 for 6 bits and 12 searches). Such a large number of horizontal wires across the entry would limit the minimum size of the CAM cell that can be achieved with this approach.

For a given search vector, the corresponding bit-wise compare lines are extended across the CAM entry and combined with AND to obtain the output match lines. Also, in these cases, the overall size of the CAM entry

is determined by the total size of match line generation circuits. The match lines are then driven outside the CAM array to enable encoder and readout circuits. These circuits would present substantial wiring and device loads, which require large match line drivers. This would also ultimately increase the CAM entry area. The wire loadings on the compare nodes (in Figure 3) would be excessive, and the device sizes should be increased to compensate for that. The CAM cell storage nodes would see increased loads as well because of the increased number of compares. This degrades the speed of the cell search and update. The cross-coupled invertors in Figure 3 should also be made larger.

All of the above indicate potential integration problems using these conventional approaches. They produce exceedingly large numbers of horizontal wires running across the CAM array and increase the overall size of the CAM entry by a large amount. Thus, these approaches are impractical in terms of chip design and make poor use of the chip area. We present a new nonconventional approach for establishing CAM and mapper organization and have developed circuit topologies that are optimal for out-of-order gigaprocessors.

A new nonconventional approach

Our approach to designing an area-efficient CAM circuit topology and to organizing the register mapper that uses the CAM array allows a fairly large number of CAM searches (12 to 16) to execute simultaneously. It offers a circuit configuration capable of maximizing the total speed of the mapper functions and enables a fast encoding of the matched registers and a fast readout of register status data array. The matched CAM entries point to the physical registers that are assigned to the logical registers presented to the mapper CAM map.

Figure 4 illustrates the key features described below. These features, unique to the present approach, provide the advantages discussed above.

- Instead of integrating the bit-compare function into the CAM circuit topology, as is the case in conventional approaches, the compare is done outside the CAM entry.
- The CAM content storage and the update portion of the CAM cell are separated from the compare component. The CAM cell contains only latches to hold the CAM-stored bits of data and a multiport multiplexer (mux) to update the CAM content. A CAM *storage entry* is then a row of these storage cells. The CAM *storage array* consists of all rows of storage entries. In the mapper application, the bits of one logical register tag are stored in one storage entry of the CAM array.
- The bit-wise compare functions of the CAM cells and the match line logic are physically separated from the CAM entry and placed in a *match entry* that is horizontally aligned with the CAM storage entry, as shown in Figure 4. The array of match entries has the same number of rows as the CAM storage array. Each search vector has its own match array. There are as many match arrays as searches to be made against the CAM stored data.
- The bits in a CAM storage entry are driven horizontally on long buses to all match arrays. There, they are compared simultaneously against all search vectors presented to the various match arrays.
- Each search vector is transmitted to a separate match array and driven vertically on long buses across the array entries. For a given match array, a match is obtained when all bits of the transmitted search vector match the corresponding bits of one CAM storage entry, which are driven to and available at the entry location of the match array. The match entry contains bit-wise compare circuits (XNORs) and a gate to AND these compares to generate a CAM match for the presented search vector.
- Each match array also contains an encoder to encode the matched physical register. The encoded physical register tag can then be transmitted to enable different circuits down the pipeline. It also contains wide muxes to enable the readout of various register status data. The status data is stored in separate arrays and is driven to the match arrays on long buses, similarly to driving the CAM storage data. Note that the match lines are used local to the match array to drive these encoder and readout circuits. This minimizes the load on the match lines and improves both the speed and the area.

The height of the CAM array is reduced by a large amount compared with the conventional CAM. The structure of the CAM entry is largely simplified by removing all of the compare circuits outside the CAM array. This alone could amount to an order of magnitude reduction in the height of the CAM cell per entry in reference to conventional CAM configurations. The actual comparisons of bit patterns occur outside the CAM array and in the match arrays. The drivers of the CAM bits to the match arrays are sized to optimize the drive delay while minimizing the CAM entry dimensions. The height of the CAM array sets up the heights of all of the remaining arrays included in the mapper and, therefore, the height of the entire mapper block. The match lines for a given search vector are generated in a separate dedicated match array. All of the match arrays receive the same CAM bit patterns (one pattern per entry), but are presented with different search bit

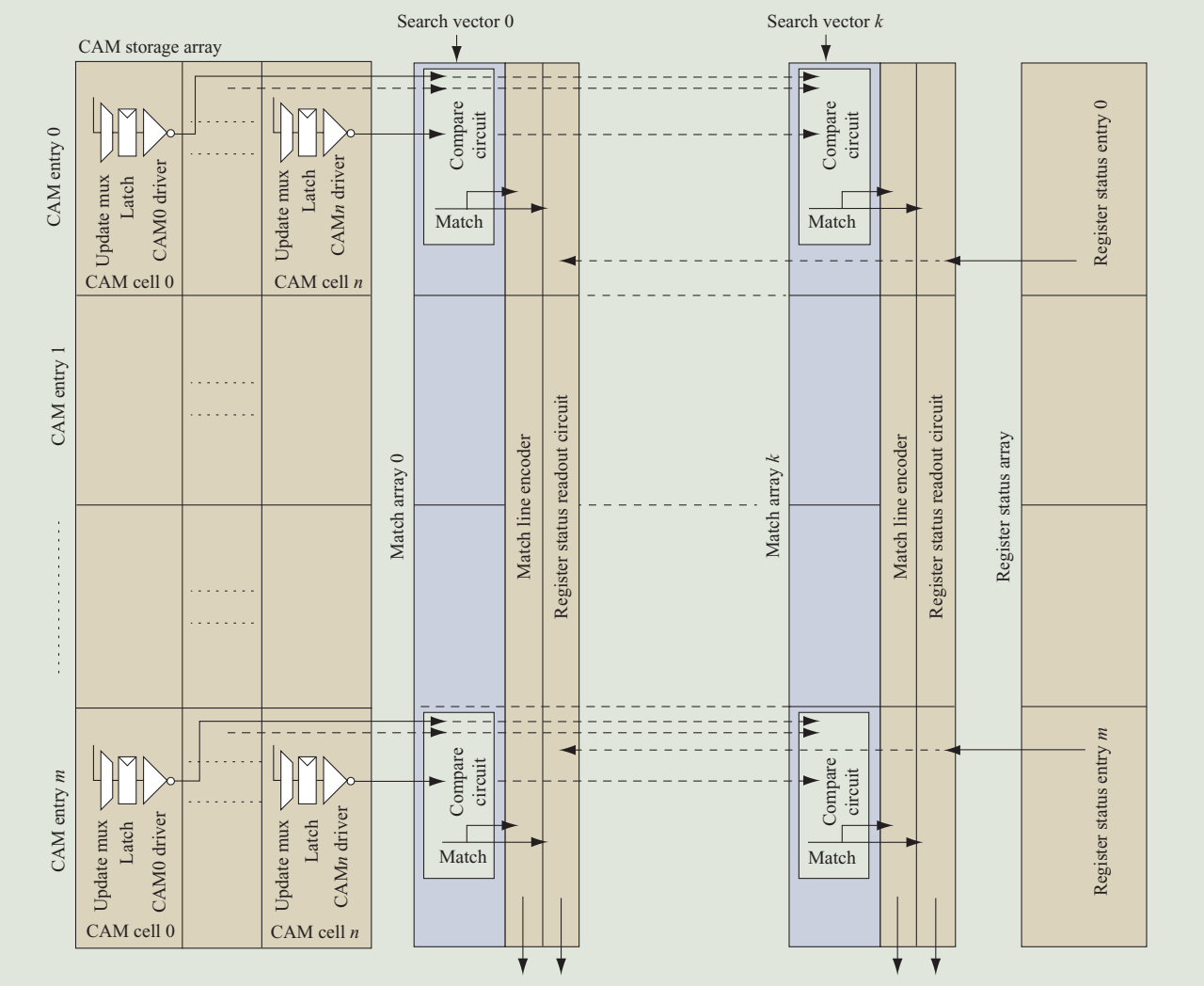

Figure 4

Register mapper organization, showing CAM storage array, match arrays, and register status ready array.

patterns. One search vector runs vertically across the entire match array to generate a match line at each entry. The match line encoder and the readout circuitry are both integrated locally within each match array and are distributed across the entire depth of the array. This is an essential feature of the organization of the mapper. It minimizes the total load on the match lines and reduces the size of the match line drivers, which, in turn, brings down the size of bit-wise compare XNORs and the overall width and area of the match array. It also increases the speed of the entire match generation and encoding by a considerable amount.

In contrast, the conventional CAM configuration discussed above presented very large loads on the match lines and incurred substantial area and speed penalties. In summary, the circuit and organization of the mapper discussed here provides a compact and area-efficient CAM and mapper floorplan that can be applied in a modern out-of-order processor chip.

Register mapper organization

The structure of the CAM and search engine described above and depicted in Figure 4 facilitates the implementation of the register mapper as a group of custom arrays. Each array is dedicated to execute a specific function. The rows of all arrays are aligned together. Each row of the array is devoted to one physical register to keep current state, shadow state, and controls for that specific register. As shown in **Figure 5**, the entire mapper is sliced into eight special custom arrays to

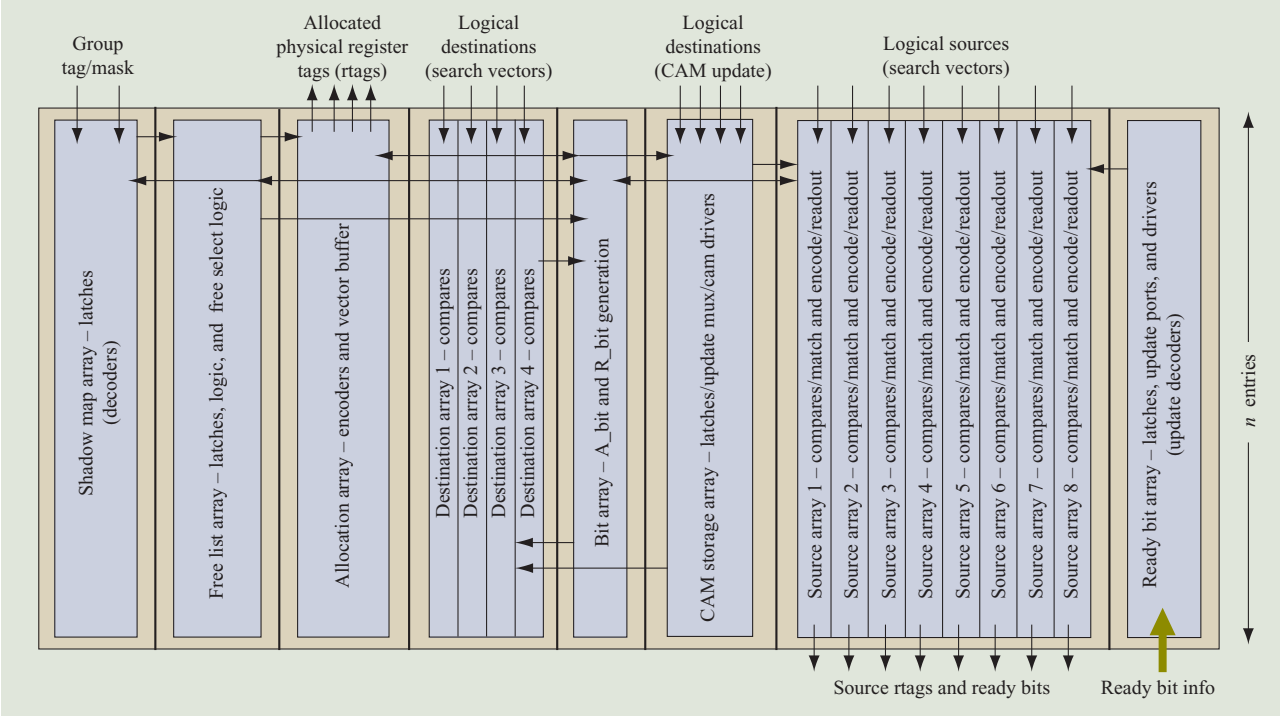

Figure 5

Overall mapper organization (eight special custom arrays).

execute the various functions of the register mapper. Figure 5 represents the mapper of the GPR, which has, in the current design, a set of eight source arrays and four destination arrays. The GPR mapper has a set of 80 physical registers, and the CAM contains a map for a total of 36 logical registers. Therefore, the CAM array has 80 six-bit-wide entries (rows), and each CAM entry stores a six-bit tag of a single logical register.

The eight mapper special custom arrays are the following:

1. The CAM storage array contains the CAM register map. Only CAM storage latches, update muxes, and data drivers are kept in this array, as explained earlier (see Figure 4). The CAM data is driven to source and destination arrays, where the actual comparisons occur.
2. The architected-bit array (bit array) holds bits to indicate an architected state (A_bit) and a replaced state (R_bit) of the physical registers. It also contains logic to generate these two bits (A and R) during instruction dispatch. These bits are used to update the A_bit and R_bit and are sent to the free list array and shadow array during subsequent cycles. After a flush, this logic also assists in restoring A_bits from the shadow maps.
3. The source array contains compare/match logic for looking up (8 to 12) logical source registers in the mapper.
4. The destination array contains logic for looking up four logical destination registers in the mapper.
5. The free list array holds bits and uses logic to generate and select a set of free registers to be used during the subsequent instruction dispatch.
6. The allocation array allocates new registers.
7. The shadow map array contains the shadow maps that are saved for each group of dispatched instructions.
8. The ready bit array has the latches to hold the ready bit register status information (w, dl0, dl1).

Circuit and logic implementation

CAM core circuit—two-cycle CAM update and drive path

The register map that is generated in the previous cycle can be used to perform the source register lookup. In this case, one has to update the CAM and drive the CAM data (the logical register tags) to the source match detection arrays at the same cycle. The match generation, the match line decode, and the register status readout must then be completed in the first half of the following

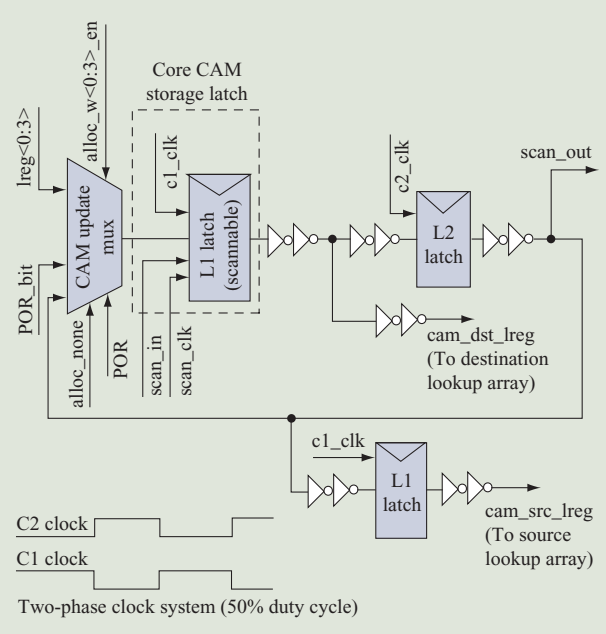

Figure 6

One-bit core CAM storage cell—two-cycle CAM update and drive path. (L1, L2: Latch whose clock signal is connected respectively to C1 clock or C2 clock.)

dispatch cycle. The mapper outputs (physical register tags and status bits) are then driven to the issue queue (ISQ) in the second half of the dispatch cycle. However, we discovered that this amount of work cannot be accomplished within the pipeline timing budget without making a cycle steal of a half cycle or more. This makes the source register lookup path extremely tight to fit within the pipeline.

To alleviate this timing constraint, for the source register lookup, we modified the CAM update and drive path to occur in two cycles instead of a single cycle. The basic idea was to add two extra latches between the core CAM storage latch and the source match detection logic, as shown in **Figure 6**. The clock system has two phases, C2 and C1. In Figure 6, L1 is a latch whose clock signal is C1, and L2 is a latch whose clock signal is C2. The first L1 latch (scannable) shown in the figure is the core CAM storage element. The L2 latch and the second L1 latch are inserted in the path to the source register lookup in order to obtain a two-cycle CAM update and drive path. To perform the source register lookup, the source arrays then use the register map generated two cycles ago, rather than one cycle ago. However, the CAM update and drive must still set up for the destination register lookup (match) on the following cycle, as shown in Figure 6. This is required by the A_bit generation logic, as is explained later.

Table 1 Rename resources of the POWER4 chip.

Mapper type	*Logical size*	*Physical size*	*Number of sources*	*Number of destinations*	*Number of ready bits*
GPR	36	80	8	4	5
FPR	32	72	12	4	3
CR	9	32	5	4	1
Link/count	2	16	3	3	1
XER	4	24	4	4	5

Because the destination match detection array is smaller and narrower than the whole source match detection array, it is possible to use a single-cycle CAM update and drive path to accomplish the destination lookup. In Figure 6, the CAM data (cam_dst_lreg) driven to the destination match detection array was taken from the first L1 latch output. In contrast, the CAM data (cam_src_lreg) driven to the source arrays is launched a cycle later from the second L1 latch. Figure 6 shows the static circuit topology for a one-bit core CAM storage cell. A 6:1 update mux is used to write the CAM cell using inputs which correspond to four destination logical register tags (lreg<0:3>) that are associated with the four instructions dispatched each cycle. The mux select signals (alloc_w<0:3>_en) are derived in the allocation array and driven to the CAM array, as described in the section on allocation logic.

When no register is allocated, the alloc_none signal (generated in the allocation logic) allows the CAM cell to keep its current state, as indicated by the feedback loop in the path. The power-on reset (POR) signal is used to initialize the CAM cell at power-on. The drivers of the CAM data to the source and destination arrays are properly sized to handle the large wire and gate loads. The first L1 storage latch is scannable, and the scan output is taken from the L2 latch output, as shown in Figure 6. For the GPR mapper, each CAM entry has six CAM bits, and the CAM array has 80 six-bit entries (**Table 1**). Physically, the CAM entries are stacked horizontally from left to right. The CAM data (cam_src_lreg and cam_dst_lreg) is driven vertically to the source and destination lookup arrays. The entries of all other arrays in the mapper are horizontally aligned with the CAM entries, as indicated earlier.

A_bit and R_bit logic and circuit

A_bit logic and circuit

The A_bit path (**Figure 7**) is constructed to satisfy two timing and machine performance requirements. The first requirement is that the A_bit cell update and drive to the source match detection array should occur in two cycles.

The A_bit becomes visible to the source match array for source lookup one cycle after the A_bit is written. This describes the normal A_bit update and drive path. Similarly to the CAM case, the A_bit update and drive must still set up for the destination register lookup (match) on the following cycle. This allows the A_bit to become visible to the destination match array immediately after it is written.

The second requirement is that the A_bit must be restored after a flush or dispatch reject. According to the first requirement, we must wait an additional cycle before dispatching after a flush recovery or dispatch reject recovery. This is highly undesirable from a performance perspective. To avoid this extra recovery cycle, we restore the A_bit at two points of the two-cycle path. We restore the A_bit at the first L1 storage latch and, at the same time, at the second L1 latch of the two-cycle path. For restore operation, this allows us to bypass the two-cycle path and instead restore the A_bit in one cycle. Instruction dispatch can then occur immediately after recovery. For normal operation, the two-cycle A_bit update path is selected to drive the A_bit to the source match array for register lookup.

Figure 7 shows the circuit path of the A_bit constructed according to the above requirements. The path starts with the logic for generating the A_bit (abit_gen), used to update the A_bit stored in the A_bit storage element (the first L1 latch). The L2 latch and the second L1 latch are inserted to obtain a two-cycle A_bit update and drive path for the source register lookup under normal operation. The bypass mux placed before the second L1 latch allows the restored A_bit to become visible to the source match detection array immediately after it is recovered and written to the A_bit entry. By adding the bypass mux, we avoided the extra cycle needed to drive the A_bit to the source array under normal operation. The delay through the bypass mux is small compared with the A_bit logic delay, and therefore there is enough time to write the restored A_bit (abit_restore) and drive the data src_abit to the source match array in a single cycle. The A_bit (dst_abit) driven to the destination match detection array is taken from the first L1 latch output in order to be available to generate a destination match that is used by the subsequent A_bit logic and update cycle.

The A_bit logic is shown in **Figure 8(a)**. The inputs to the A_bit logic are derived in various arrays and logic macros inside and outside the mapper block. The dst_match is generated in the destination array to indicate a destination match (see the section below on destination match). The dispatch valid signal (disp_valid) is derived in the dispatch logic macro (see the dispatch section below). Both signals are driven to the A_bit array and locally latched using L2 latches. The "i" denotation in

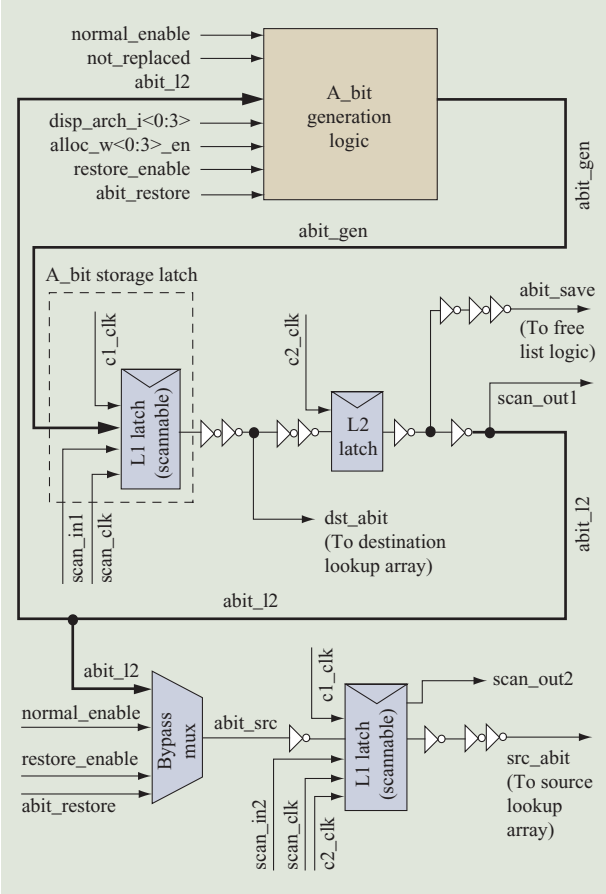

Figure 7

A_bit logic and circuit path.

Figure 8 refers to the *i*th entry of the A_bit array. The restore_enable_shadow is derived in the shadow map array and driven to the A_bit. The array_init is a mapper initialization signal. Both signals latched local to the A_bit using master–slave latches (L1/L2). The disp_reject signal is created in the dispatch logic macro (located outside the mapper—see the dispatch section) and then driven to and locally latched in the A_bit array. The input signals disp_arch_i<0:3> are the architected states associated with the four instructions that are dispatched in one cycle. They are created by the dispatch logic (see the dispatch section) and then driven to the A_bit array and latched locally using L2 latches.

Each of these signals runs horizontally across the mapper, driving all of the entries in the array. The alloc_w<0:3>_en inputs are the allocation write-enable signals originated in the allocation array and driven directly to the A_bit entries. The same signals also feed the CAM array and the R_bit array, and run vertically along the mapper. The input abit_src to the second L1

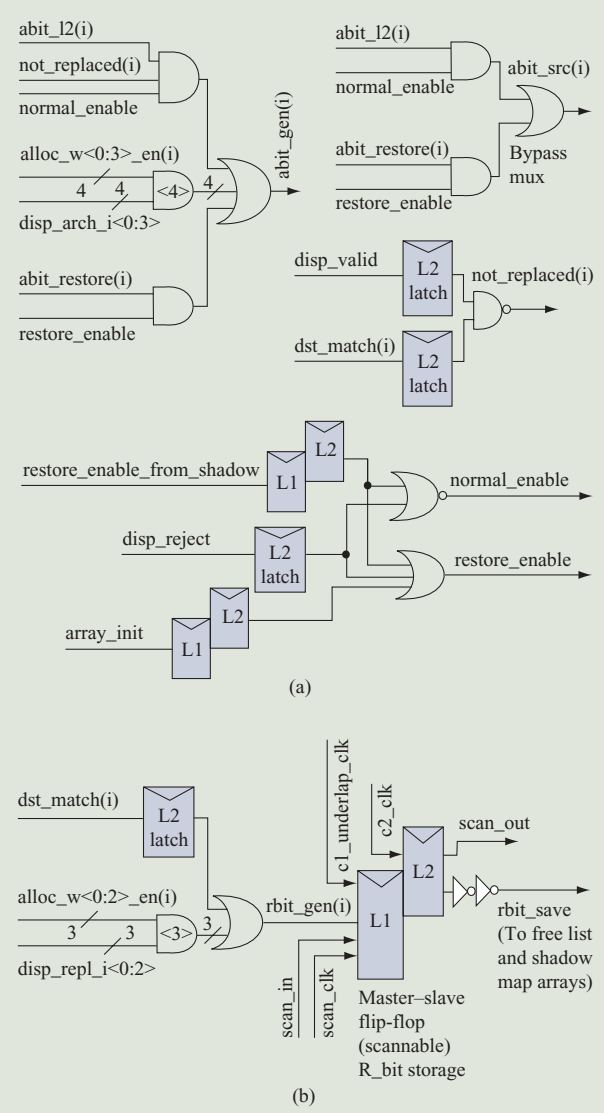

Figure 8

(a) A_bit logic and (b) R_bit logic and storage.

latch in Figure 7 is selected by the bypass mux in Figure 8(a). The abit_restore(i) are the A_bits that are restored from the free list array and driven directly to the A_bit logic. The abit_l2(i) are the current A_bit entries that are held by the L2 latches of the two-cycle path of Figure 7. The second L1 latch is also scannable to permit testability of the bypass mux. The A_bit output data src_abit and dst_abit are driven vertically to the source and destination lookup arrays, respectively. The outputs abit_save in Figure 7 are sent vertically to the free list array and used there as inputs to the free list bit logic. One single A_bit is assigned for each entry in the A_bit array. For the GPR mapper, the A_bit array contains 80 one-bit entries (Table 1), with 80 corresponding to the number of physical registers within the register file managed by the mapper.

R_bit logic and circuit

Registers are *replaced* when a subsequent instruction writes to the same logical register target. Some registers may be replaced by subsequent instructions within the same group and are allocated and replaced at the same time. The replaced status of the registers in the mapper is maintained in the replaced bit (R_bit) array. The R_bit array contains logic for generating and updating the R_bit and also contains storage elements to store the current R_bit values. **Figure 8(b)** shows the R_bit circuit path. A scannable master–slave flip-flop latch (L1/L2) is used to store the R_bit. To prevent a flush-through problem, the underlapped C1 clock is used to clock-gate the master latch. The rising edge of the underlapped C1 is delayed by a determined amount relative to the rising edge of C1. The slave latch L2 is clock-gated by the C2 clock. The outputs [rbit_save in Figure 8(b)] are driven vertically to the shadow map array to be saved there during the cycle after dispatch. The same R_bit data also feeds the free list array and is used as inputs to the free register generation logic. The inputs to the R_bit logic are dst_match, disp_repl_i<0:2>, and alloc_w<0:2>_en. The dst_match signals are driven vertically to the bit array to feed both the R_bit and the A_bit arrays. The input signals disp_repl_i<0:2> are the replaced states associated with instructions 0, 1, and 2 that are dispatched in a single cycle. They are created by the dispatch logic (see the section on dispatch) and then driven to the R_bit array and latched locally using L2 latches.

Each of these dispatch signals runs horizontally across the mapper and drives all of the entries in the array. The alloc_w<0:2>_en inputs are the same allocation write-enable signals that drive the A_bit (and the CAM) entries. The resulting rbit_gen(i) is used to update the R_bit *i*th entry. One single R_bit is assigned for each entry in the R_bit array. For the GPR mapper, the R_bit array has 80 one-bit entries (see Table 1).

Source lookup

The source lookup circuits of the source arrays contain compare and match logic for looking up the logical source registers in the mapper. The logical source registers for the entire instruction set dispatched in a single clock cycle are looked up simultaneously. For the GPR mapper, eight source searches are done in parallel during a single dispatch cycle. For the FPR mapper, 12 source registers are searched in the same dispatch cycle. According to this, there are eight separate source arrays in the GPR mapper, while the FPR mapper contains 12

separate source arrays. The same source array macro is instantiated multiple times. However, each source array is fed by a different logical source register index associated with the dispatched instruction set. In contrast, the same CAM bits in a given entry drive all of the source arrays placed in a given mapper, as explained earlier in the CAM core circuit section. The logical source register index (up to 6 bits) is compared with the contents of each CAM entry (up to 6 bits), and a match line is generated for each entry (physical register) in all of the source arrays of the mapper. A match occurs if all of the CAM bits of the same entry are equal to the corresponding bits of the logical source register index, and, at the same time, the architected bit (A_bit) of the entry is "1" (the corresponding register is in the architected state).

Figure 9 shows the general timing of the mapper functions associated with dispatch. The *transport* cycle is the cycle before the *dispatch* cycle, during which instruction information is transported from the instruction decode unit (IDU) to the dispatch buffer. The logical register indices are transmitted during the transport cycle from the dispatch buffer and received by L1 latches local to the mapper. The source lookup begins during the second half of the transport cycle when the logical source indices (Lregs) are launched from the mapper L1 latches. In the current floorplan (oriented 90° relative to Figures 4 and 5), the Lreg L1 latches are placed at the left-side or right-side perimeter of the source arrays. The same Lreg index is then driven horizontally across the mapper source array to feed the bit-wise compare logic of all of the entries in the mapper (a fan-out of 80 in the case of the GPR mapper). As described earlier in the CAM core circuit section, the CAM bits from each CAM entry are also launched from L1 latches located in the CAM array macro and are then driven vertically to feed the bit-wise compare logic for all of the source arrays placed in the mapper. Similarly, the A_bit is transmitted from the L1 latches in the bit macro and driven vertically to the match generation circuit locations of all of the source arrays in the mapper. At the intercept points of Lreg bits and CAM bits, simple pass-gate XNOR circuits are used to do a bit-wise compare between the corresponding CAM and the Lreg bits. An entry match line is obtained by ANDing together all of the single-bit compares and the A_bit of that entry.

Figure 10 is a circuit diagram of the source lookup pathway illustrated for a single mapper entry. In each source array, the match lines are encoded using a local highly distributed encoder circuit to derive the physical register tags (rtags). The match lines are also used to read out the register status bits (w and dl bits) using distributed wide muxes local to the source array. The w and the dl bits are generated in the ready bit macro and driven vertically to all of the source arrays in the mapper (see the ready bit circuit in the section on the ready bit array below). The rtags and the register status bits for each source array are latched locally by L2 latches to give the main outputs of the mapper. These mapper outputs are then driven outside the mapper to the dispatch select logic and propagated through the issue queue macro muxes to the receiving master–slave (L1/L2) latches. The mapper source output is read from the mapper during the first half of the dispatch cycle (see Figure 9), allowing time to go through the dispatch select logic and mux and write the data into the issue queue.

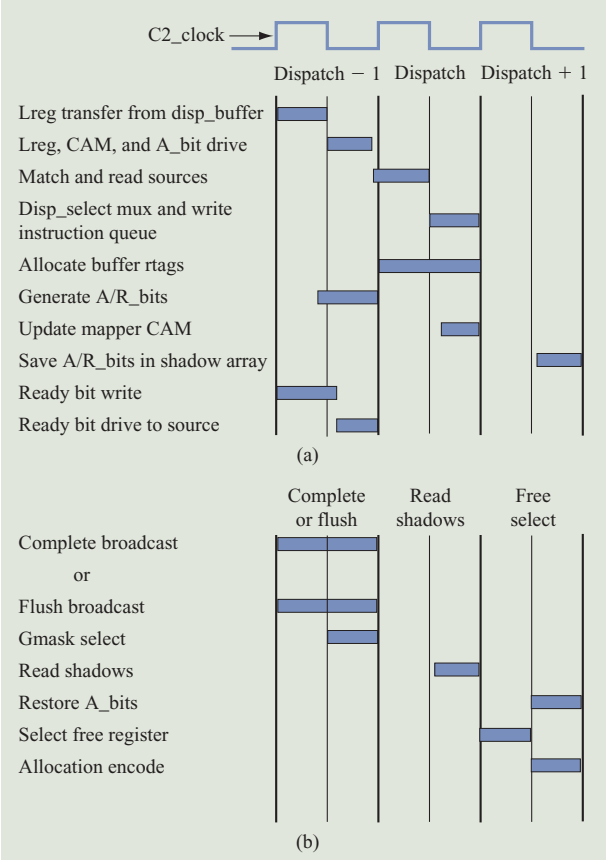

Figure 9

General timing for mapper functions associated with (a) dispatch (lookup and update) and (b) completion, flush, and reallocation.

Destination lookup (destination match)

The destination lookup and match is performed to detect registers that are replaced and whose A_bits must be cleared. For each entry in the mapper, four logical destination registers are compared with the contents of the CAM, and four match lines are produced. The four logical destination registers correspond to the destinations of the four instructions dispatched in a single

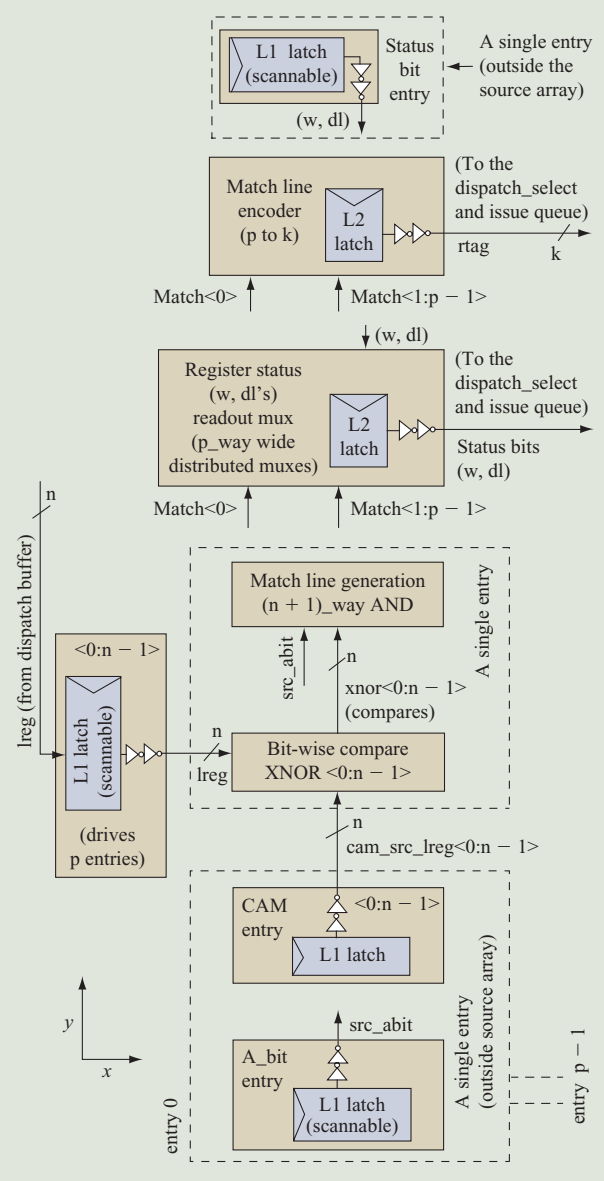

Figure 10

Source lookup circuit pathway (illustration for a single entry and a single source array). [n: number of CAM bits per entry; p: number of entries (physical registers); k: number of bits in rtag index.]

cycle. These four match lines are then ORed together to generate the destination matches (dst_match) as outputs. The dst_match signals are driven to the A_bit array to be used in updating the A_bits and R_bits, as described earlier.

The CAM and A_bit update and drive paths are set up for destination lookup on the following cycle. This allows the CAM bits and the A_bit to become visible to the destination array immediately after they are written. The circuit pathway used to obtain the destination match is similar to the one used in the source lookup. A destination match occurs if all of the entry CAM compares are "1," the entry register is architected, and the destination is valid (dst_valid is "1"). The destination logical registers (lregs) are launched from L1 latches placed local to and at the left-side or right-side perimeter of the destination arrays. The same lreg index is then driven horizontally across the mapper destination array to feed the bit-wise compare logic of all of the entries in the mapper. The lreg wires run orthogonal to the CAM bit wires. The destination lreg signals are also driven vertically along the array perimeter to the CAM macro to be used to update the CAM array.

The dst_valid input signals are derived in the dispatch buffer logic and transmitted to and captured by L1 latches located at the perimeter of the destination macro. The dst_valids are then launched from the L1 latches and run horizontally to feed the match circuits. Four similar match arrays are used to give four match lines per entry. Each match array is fed by a corresponding set of an lreg index and dst_valid. For each entry, the four match lines are ORed together to obtain the entry destination match signal (dst_match). The dst_match output signals are run vertically to the neighboring A_bit/R_bit macro.

Free list generation and selection of free registers

The architected, held, and buffered bits (A, H, and B bits) are kept in the *free list* macro to indicate the current state of each register in the mapper. This set of register-state bits is used to generate a free register list. These bits are updated during the cycle after dispatch, using information from the bit macro; during the cycle after a completion or a flush signal is received, using information from the shadow map macro; or during the free select cycle. The free list macro (**Figure 11**) contains write logic to set or clear each of the state bits (A, H, and B). It also contains a simple logic to create a *free* bit for each register based upon the register state bits and whether the register is already a selected free register. In addition, the free list macro contains a free select logic that selects a single free register from each mapper quadrant as a candidate register to move into the allocation buffer.

For the GPR mapper of 80 registers and four quadrants, the select logic selects one register from 20 per quadrant. The selected free registers are preallocated into an allocation buffer to provide early information for future dispatches. The free list bit logic and macro floorplan are illustrated in Figure 11. The free list select logic is shown in **Figure 12(a)**. Two clock cycles are used to fit the entire pathway. The first clock cycle is used to update the A, H, and B bits (AHB). Master–slave L1/L2 latches are used to store the data. The second clock cycle is used to generate free registers and then carry out

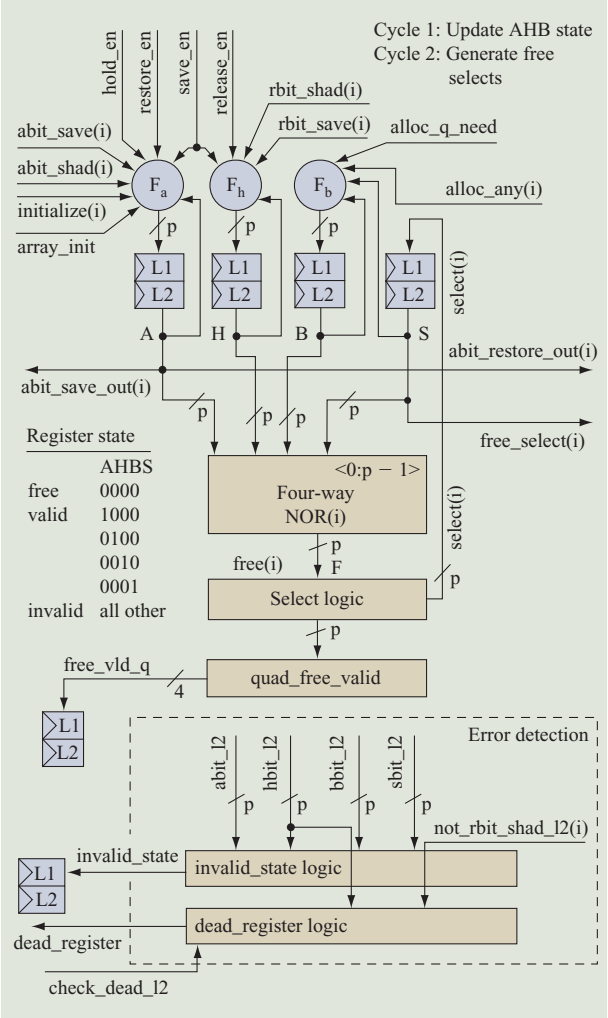

Figure 11

Free list AHB register state logic and macro floorplan (two-cycle pathway). F_a, F_h, and F_b represent the update logic described in Figure 12(b). [p: number of entries (physical registers); i = 0 to p − 1.]

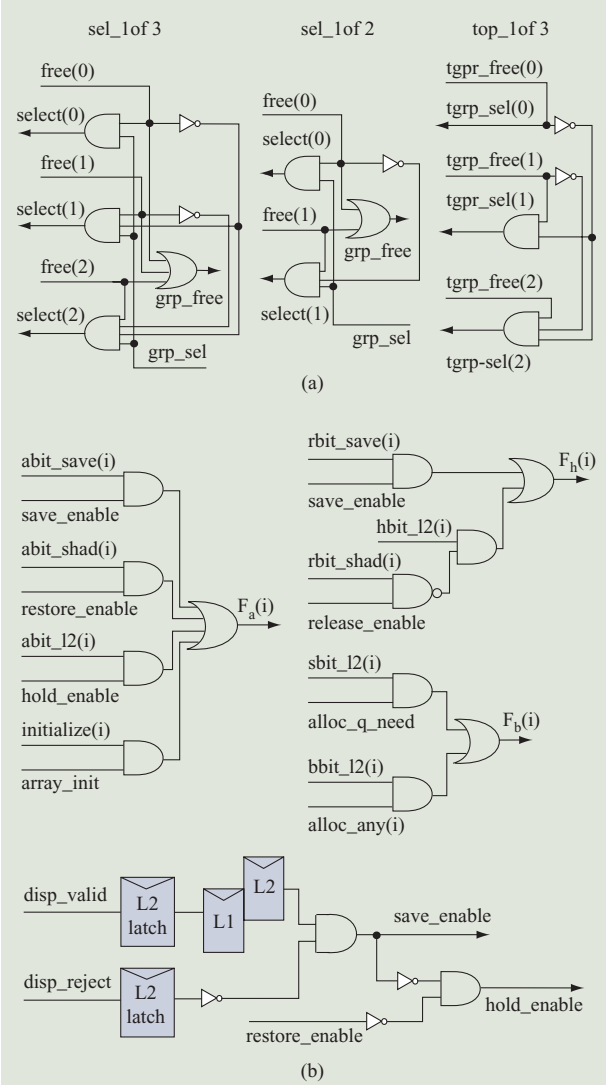

Figure 12

(a) Free list select logic; (b) free list bit logic.

the select logic cycle to select a single free register per quadrant for preallocation. An S_bit is kept in the free list array to designate the *selected free* state of the register. S_bits are also stored in L1/L2 latches and then driven outside the free list macro to the allocation array to feed the allocation buffer. The S_bits generated in the previous cycle (sbit_l2) are used locally to update the B_bit, as shown in **Figure 12(b).**

The F_a, F_h, and F_b (in Figure 11) represent the update logic described in Figure 12(b). The abit_save and rbit_save come from the bit macro, while the abit_shad and rbit_shad signals come from the shadow map array. The alloc_q_need and alloc_any signals are driven from the allocation macro. The abit_l2, hbit_l2, bbit_l2, and sbit_l2 are the stored A, H, B, and S bits, respectively. The restore_enable and release_enable come from the shadow map array. The other enable signals (save_enable and hold_enable) are generated locally. The disp_valid and disp_reject signals are originated from the dispatch buffer logic. The index "i" indicates the *i*th entry. A free register is the NOR of A, H, B, and S_bits (AHBS). A state register of AHBS = 0000 gives a free register.

The selection cycle for the case of the GPR mapper with 20 registers per quadrant selects the top single free register in one quadrant. The sel_1of3, sel_1of2, and top_1of3 logic functions are described in Figure 12(a). The sel_1of3 selects first one free out of three, and sel_1of2 selects the first free out of two. The top_1of3

selects the top group that has a free register. For each quadrant, a free valid signal (free_vld_q) is generated to indicate whether at least one register is free in that quadrant. These signals are used in the allocation macro. The free list macro also contains error detection logic to diagnose invalid states and dead registers. AHBS states other than 0000, 1000, 0100, 0010, or 0001 are invalid. A dead register check is made to detect a register that is stuck in the hold state (hbit_l2 mismatches the R_bit saved in the shadow map).

Allocation of registers—encoded and vectored allocation buffers

In each cycle, up to four registers may be selected and moved from the free list (free macro) to the allocation buffer (allocation macro). The preallocated registers provide early information for future dispatches. New registers are allocated during dispatch for each instruction operation that writes a new result. Two versions of the allocation buffer are kept in the allocation macro. The *encoded allocation buffer* keeps encoded register tags (indices) that must be sent to local and remote instruction issue queues during dispatch. The *vectored allocation buffer* keeps the same register identifier in a vectored format (non-encoded) that is used to control register allocation and CAM map update within the mapper during the dispatch cycle. The allocation macro also contains a logic to derive signals used to keep track of register availability and to generate a *resource hold* signal for the mapper when necessary. In the current floorplan, the encoded allocation buffer and logic are placed in the perimeter side of the allocation array. The vectored allocation buffer occupies the allocation array.

Encoded allocation buffer

The encoded buffer receives the free select signals (free_select<0:p–1>, where *p* is the number of registers in the mapper) from the free list macro described in the free list generation and selection of free registers section above. It also receives valid bits from the free list macro. The register allocation is done on a quadrant basis, with up to one register allocated from each mapper quadrant in a single dispatch cycle. In line with the free list macro, this buffer is also partitioned into four quadrants (except for the CTR/LR mapper, which has three triplets). In each quadrant, the set of free_select signals is encoded to generate one register tag per quadrant.

As shown in **Figure 13**, the encoded buffer contains a set of encoders to read out a register tag from each quadrant of the mapper during the second half of each cycle (L1 phase). At the same time, it reads out a valid bit from each quadrant. The entire set of register tags in each quadrant and valid bits from all four quadrants are sent through 4:1 muxes (7-bit, for GPR and FPR mappers), followed by 2:1 muxes (7-bit), and then captured by L1 latches that feed into the output L2 latches.

The four 4:1 muxes (one 7-bit, 4:1 mux per quadrant) perform rotating writes to the four allocation buffer slots that correspond to the four instructions dispatched in a single cycle. A 4-bit counter is used to provide the same four select signals for all of the 4:1 muxes of the four slots. In Figure 13, the inputs rtag0<0:6>, rtag1<0:6>, rtag2<0:6>, and rtag3<0:6> to the four allowed buffer slots are rotated such that each slot of the buffer is written by a different quadrant during the four consecutive cycles of the counter, and the input rotation is such that in each of the counter four cycles, the four slots receive their tag inputs from different mapper quadrants in a rotating manner. This guarantees that each quadrant of the mapper free list writes to alternating slots of the allocation buffer during four subsequent dispatch cycles. This rotation increases the likelihood that an allocation slot that remained empty in a given dispatch cycle will be filled in the next cycle.

For the next instruction dispatch to occur, all needed allocation slots must be filled and the required free registers from the designated quadrants must be available. An allocation slot is filled with a new rtag if allocation is requested by the dispatch block (disp_i<0,1,2,3>_alloc is asserted) or if the slot has an invalid allocation in the previous cycle (alloc_valid_l2_b is asserted). A slot fill signal, tag_fill_s, is derived locally, as shown in Figure 13, to update the slot allocation buffer. Only slot0, which allocates the destination register tag for instruction i0, is fully illustrated in Figure 13. Register tags are similarly allocated for instructions i1, i2, and i3 by rotating the same inputs to the slot 4:1 muxes. The alloc_rtag and rem_alloc_rtag are sent to local and remote dispatch select macro and remote mappers. The outputs zero_free<0:3> deliver an early L1 signal to the dispatch control logic for computing the dispatch reject.

Vectored allocation buffer

The vectored allocation buffer keeps the same register identifier in a vectored format (non-encoded). It provides the write-enable signals that are used to update the CAM logical-to-physical register map within the mapper during the dispatch cycle. These write-enable signals are also used to update the A_bit and R_bit, as explained earlier.

The free select signals from all quadrants are muxed and latched to provide the write-enable signals alloc_w0_enable<0:79>, alloc_w1_enable<0:79>, alloc_w2_enable<0:79>, and alloc_w3_enable<0:79>, as shown in **Figure 14**. These enable signals are driven to the CAM array to update the CAM entries through the four write ports of the CAM cells. The same signals also drive the A_bit and R_bit logic. Each instruction that

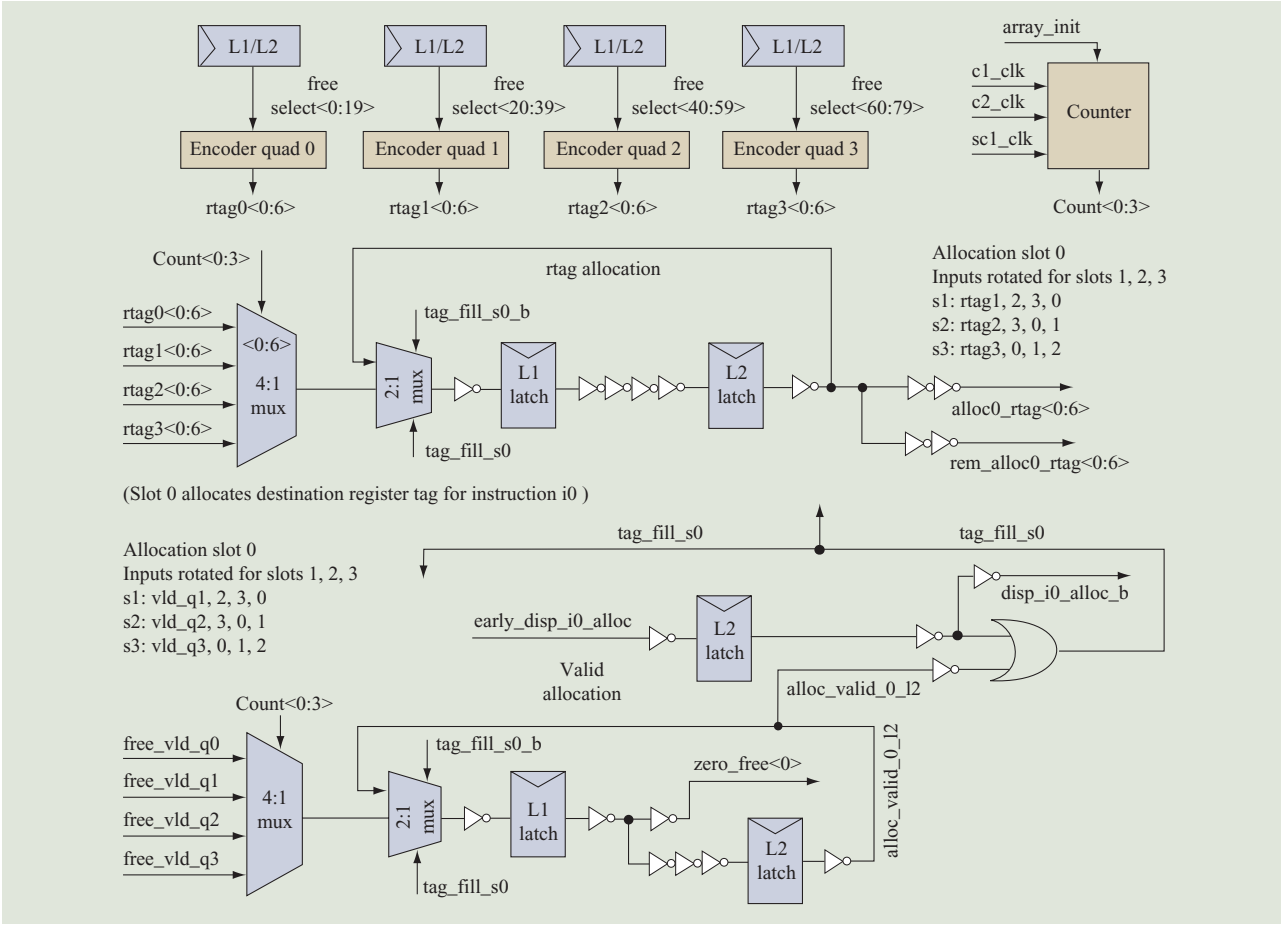

Figure 13

Encoded allocation buffer (only slot 0 is illustrated; the same inputs to the 4:1 muxes are rotated for slots 1, 2, and 3).

writes a result will update a single CAM entry during the dispatch cycle. The vectored allocation buffer slots are filled in a manner consistent with the encoded allocation buffer, as shown in Figure 14.

For example, the mux select signal q0_take0 allocates a destination register for instruction i0 from quadrant 0 during count 0 of the allocation counter and provides alloc_w0_enable<0:19>. In count 0, instructions i0, i1, i2, and i3 get their allocations from quadrant 0, 1, 2, and 3, respectively. In count 1, instructions i0, i1, i2, and i3 get their allocations from quadrant 1, 2, 3, and 0, respectively, and so on for count 2 and count 3. This is consistent with the filling of the encoded allocation buffer with register tags. Allocation-from-quadrant-needed signals alloc_q0_need, alloc_q1_need, alloc_q2_need, and alloc_q3_need are generated as shown in Figure 14 to signify that allocations were needed from the corresponding quadrant (q0, q1, q2, and q3). These signals are sent to the free list logic to define the B_bit of the mapper register state. A register is moved to the buffered state if it is selected from the quadrant free list and allocation is needed from that quadrant in the same cycle. The alloc_any signals are also sent to free list logic to reset the B_bit. The alloc_none signals are driven to the CAM array to rewrite the CAM entries.

Shadow map array

The mapper not only keeps the current state of the registers, it also maintains shadow maps to save previous mappings and replacement information for each group of instructions that has been dispatched but not yet completed or flushed. Two bits, the A_bit and R_bit, are saved for each of the 20 groups of dispatched instructions that are currently active and for each register in the mapper (80 for the GPR mapper). These bits are stored in two static random access memory (SRAM) arrays with one read port and one write port per bit. For the GPR mapper, two 20-bit × 80-row arrays are used. These two

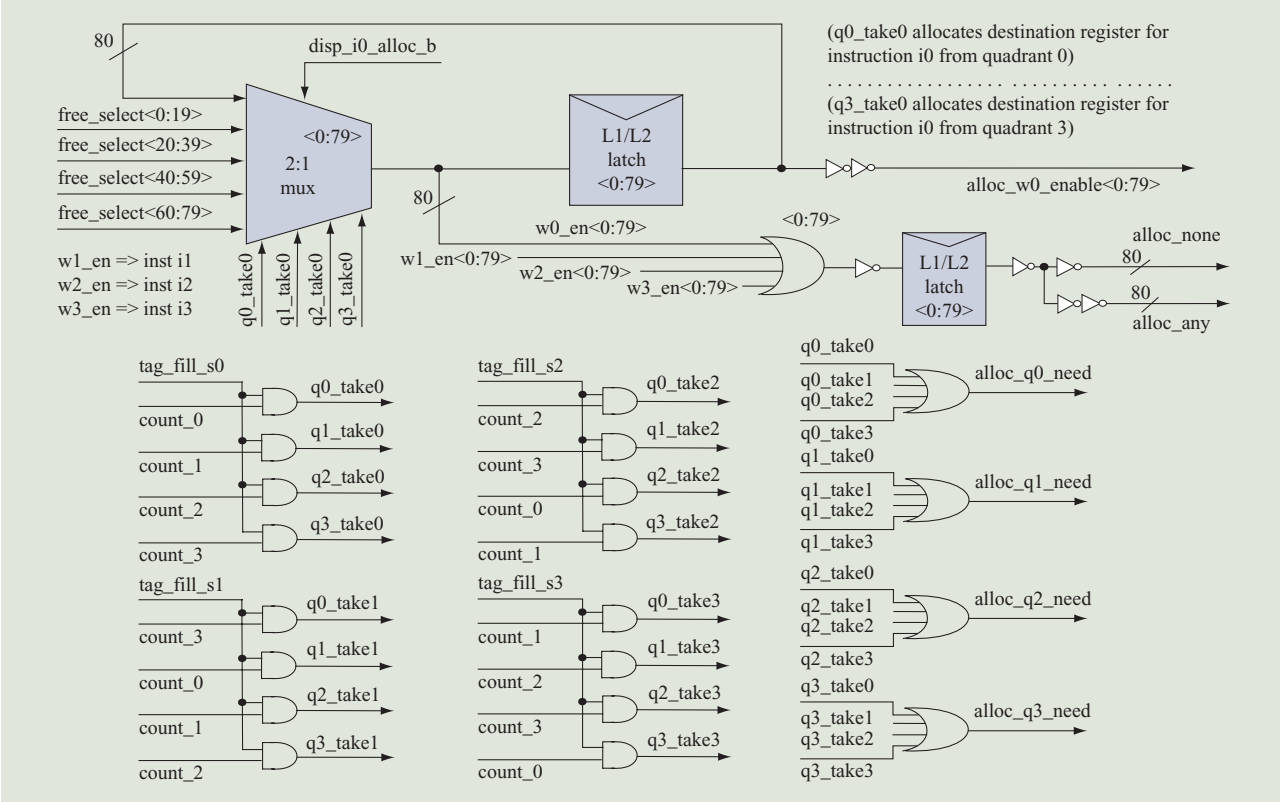

Figure 14

Vectored allocation buffer. (Only slot 0 is fully illustrated; it allocates destination registers for instruction i0. Instructions i1, i2, and i3 are allocated similarly.)

arrays are vertically aligned with the rest of the mapper arrays with word lines (rows) that run horizontally and bit lines that run vertically. The bits are written during the cycle after dispatch from latches that are placed in the A_bit/R_bit arrays, described above in the section on A_bit and R_bit logic.

The A_bit and R_bit logic formulates the A_bits and R_bits needed to update the current register map and save a shadow copy of these two bits to be stored in the shadow map memory array. The shadow bits are read during the cycle after instruction completion or instruction flush is received. The two events, instruction group completion and flush, cause registers to be released or restored. When a group completes, the results that it produced are committed, and the registers that it replaced are released to the register free list. When a group is flushed, the registers that it allocated are released. Additionally, when a flush occurs, the registers that were in the architected state just prior to the first flushed group are restored to the architected state. In this way, the proper state is restored.

Figure 15 shows the shadow map arrays and control logic. A dispatch group tag (alloc_gtag) is provided during the dispatch cycle and must be decoded, then used to control a write into the shadow maps on the subsequent cycle. A vector of A_bits and R_bits (abit_save and rbit_save) are saved into the array for each dispatch group. This write must be prevented in the case in which a group was dispatch-rejected. Completion and flush signals (comp_valid and flush_valid) are received and latched here; then restore_enable and release_enable signals are sent to other macros on the subsequent cycle. The restore_enable signal must come on after a flush. The release_enable signal must come on after a completion or flush. One mask (flush_gmask) is used to indicate one or more groups being flushed when a flush is valid (flush_valid). We must read out and summarize the R_bits for all groups being flushed.

Another mask (misc_pmask) is used to indicate the group being completed when completion is valid (comp_valid). We must read out the R_bits for the group being completed. This second mask is also used to indicate the oldest group being flushed. We must read

out the A_bits for the group being flushed. There is one possible hazard that occurs if the completion logic indicates a valid flush with a flush_gmask of all zeros! This condition is not allowed to occur.

Ready bit array

Special ready bits (w, dl0, dl1) are kept in the ready bit array for each register to indicate whether its data is available (possibly conditioned upon the outcome of a load) **(Figure 16)**. Each instruction source has a w bit to indicate whether its data is ready (written). The two-bit dl0 and dl1 track instruction dependence on load for load-store units ls0 and ls1. If any of the two bits in dl0 or dl1 are set, it indicates that the corresponding source operand directly or indirectly depends on a load that is in progress, but the cache hit/miss (ls0, l_data_valid in Figure 16) is not yet known. This instruction can be issued before we know whether the cache hit, but must be reissued if a cache miss occurs. The instruction issue logic provides this information when a result is about to be written into the register file. Only ready bits written during the previous cycle are available during dispatch.

As shown in Figure 16, the array includes latches to hold the ready bits (w, dl0, dl1). These latches require some special shifting and resetting functions. Split L1 to L2 latches are used to store, drive, and shift the ready bits. The ready bit data (W, DL0_b0, DL0_b1, DL1_b0, DL1_b1) is driven to the source array and is available to be read out for each source. The source macro sends the read information to the issue queue during dispatch. Update decoders are also required to control update of these bits as register values become available. When a register is allocated, its w and dl bits are cleared. When a register value becomes available, the w bit is set to 1 and the dl bits are set using information derived locally (wrt_ls0 and wrt_ls1) from inputs provided by the issue logic. Otherwise, the dl bits shift logically each cycle toward bit 0. The shifting of dl bits is done before they are written into the mapper for each cycle they are in the mapper latches and after they are read out of the mapper.

The ready bit reset logic is also shown in Figure 16 using inputs from the issue queue. The two load-store units provide the data valid signals (ls_data_valid). Master–slave L1/L2 latches are used to hold these inputs and also to hold the update decoder vectors. *Reset* overrides any other functions and occurs for allocation reset, load miss, and power-on reset.

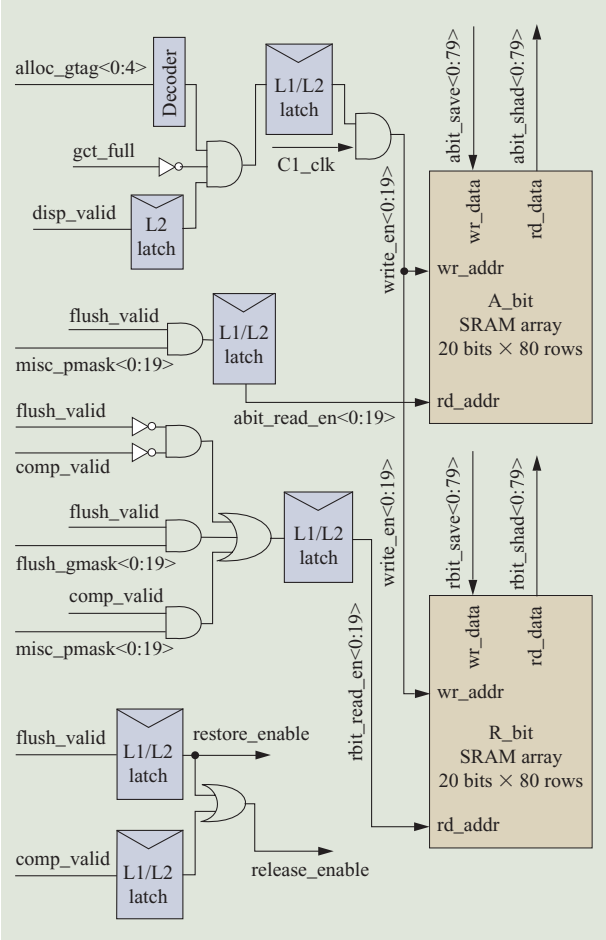

Figure 15

Shadow map arrays and control logic for the shadow of the GPR mapper of 80 registers.

Dispatch

Instruction sequencing unit

The instruction sequencing unit (ISU) manages out-of-order instruction execution within the processor core [10]. It implements the following microarchitectural functions:

- *Dispatch:* Described below in this section.
- *Register renaming and allocation (mapper):* The topic of the present paper.
- *Issue:* It queues instructions, monitors dependences, and controls out-of-order instruction issue to the fixed-point unit (FXU), load-store unit (LSU), floating-point unit (FPU), and branch unit (BRU) [10].
- *Completion:* It monitors the finish status of each instruction, ensures in-order instruction completion, controls resource deallocation, and initiates selective instruction flushes when necessary [10].

These functions are performed during different stages of the pipeline [10], and each function requires special structures and logic.

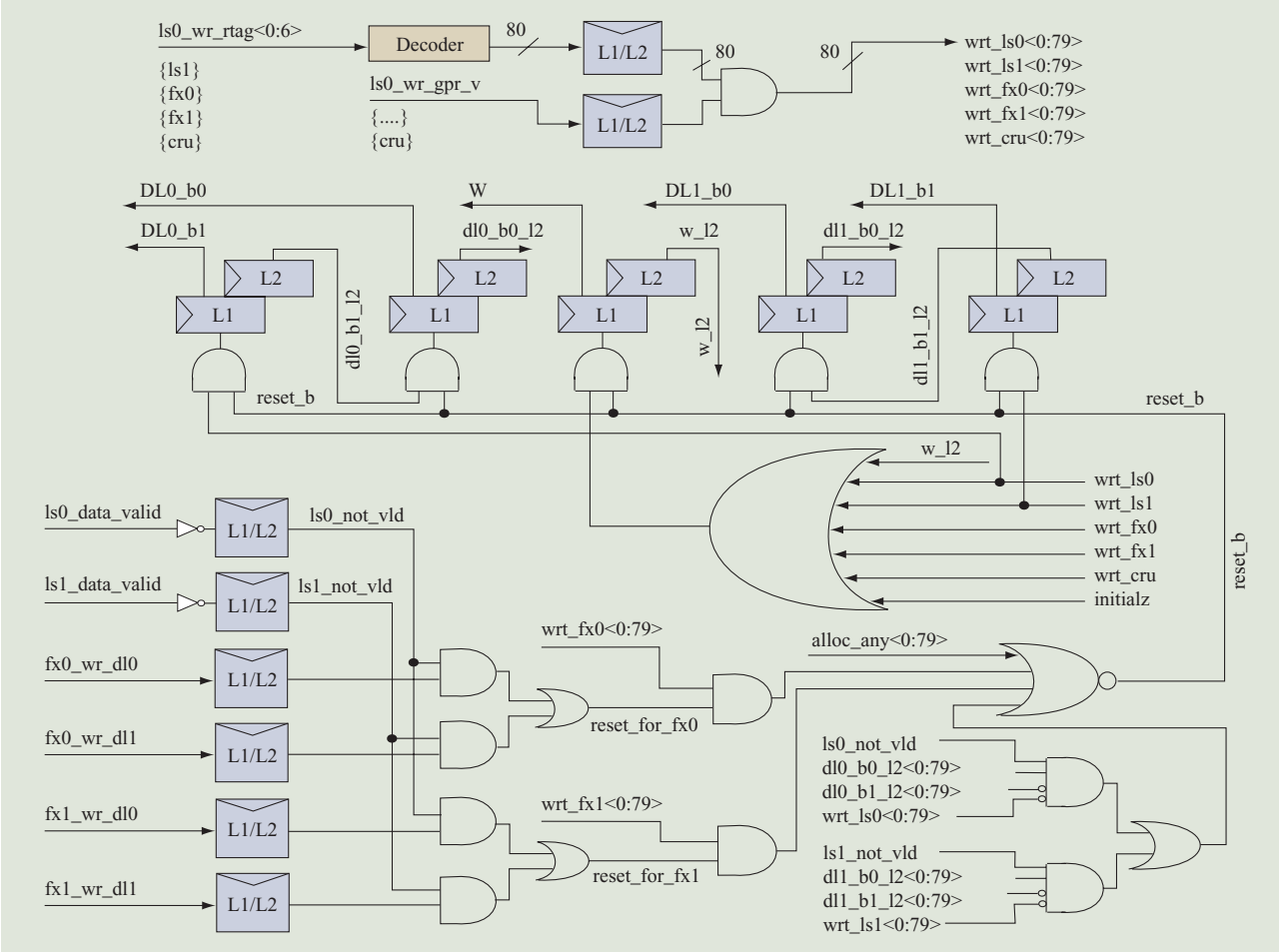

Figure 16

Ready bit array and control logic (one entry of the GPR mapper of 80 registers is shown).

Dispatch

The dispatch buffer and control logic receive groups of instructions (iops) in order from the instruction decode unit (IDU) and dispatches them or holds them when necessary. Instructions are grouped and aligned by the IDU such that no additional alignment is necessary. Before a group can be dispatched, all resources required by the group must be allocated by the ISU. The ISU must allocate many different resources during dispatch. The group completion table entry, issue queue slots, rename registers, load reorder queue entries, and store reorder queue entries must be assigned before successful dispatch [10]. The dispatch logic determines ahead of time whether resources will be available and precomputes a dispatch reject signal when necessary. The dispatch engine performs many functions:

- *Allocate group tag:* Up to five instructions may be dispatched during a cycle (the fifth one may only be a branch). These instructions are assigned a group tag, or *gtag*. This tag is used to selectively serialize or cancel instructions as they proceed down the pipeline. It also corresponds to one entry in the completion table in which information about each group is maintained. The completion logic provides the next available gtag, and the dispatch logic indicates allocation.
- *Allocate load/store tags:* Each load instruction is assigned a load tag, or *ltag*. Each store instruction is assigned a store tag, or *stag*. An ltag and an stag flow down the pipeline with each load or store instruction to indicate load/store ordering information. These tags correspond to load and store reorder queue entries in the LSU. The completion logic provides

the next four available ltags and stags. The dispatch select logic must identify load/store instructions and select an appropriate tag for each load and store within a dispatched group.

- *Move instructions to issue queues:* The dispatch dataflow and control logic is centralized to feed each of the four major ISU partitions, as shown in **Figure 17**. This includes the setup/overflow registers to receive transmitted instructions from the IDU, the dependency comparison blocks, the dispatch/reject registers, and all associated dispatch control logic. The IDU instruction buffer and instruction bus feed the dispatch buffer. Instruction groups are dispatched into the appropriate issue queues one group at a single cycle [10].
- *Determine intragroup dependences:* Instructions dispatched at the same time may be interdependent. Register dependences must be identified by examining decoded source and target register information and comparing the appropriate register indices. The IDU provides some partial dependence information, and the dispatch logic performs the final dependence detection. Dependence information is needed to generate register allocation and selection controls.
- *Floating-point status control register* (*FPSCR*) *handling:* An entry in the FPSCR result buffer is allocated for each group of dispatched instructions. The gtag of the last group to do a general write to the FPSCR must be passed along as a special rtag for the floating-point instructions. Additionally, all instructions in a group must be analyzed, and the last one to write the FPSCR result flags must be identified and marked.
- *Miscellaneous decode:* Some information may have to be decoded and detected and passed along to the issue queues and execution units. Multicycle operations, for example, may be detected during dispatch. A scoreboard interlock must also be managed to enforce ordering for iops that read and write non-renamed resources.

The dispatch setup cycle is one cycle before dispatch. Instruction groups come from the IDU validated by a signal group_valid. The groups are latched into a master–slave L1/L2 stage in a dispatch setup macro. This latch stage is one cycle before dispatch, and in this cycle we perform functions such as current-cycle dependency compares, previous-cycle dependency compares, and setting up the early dispatch outputs to the mapper. These outputs consist of the mux signals, which are L2 phase signals representing what the lreg and valid bits will be for dispatch_valid in the next cycle (the dispatch cycle). Also, the early (or L1-launched) allocation and replace signals are launched from the dispatch L1 stage. During the

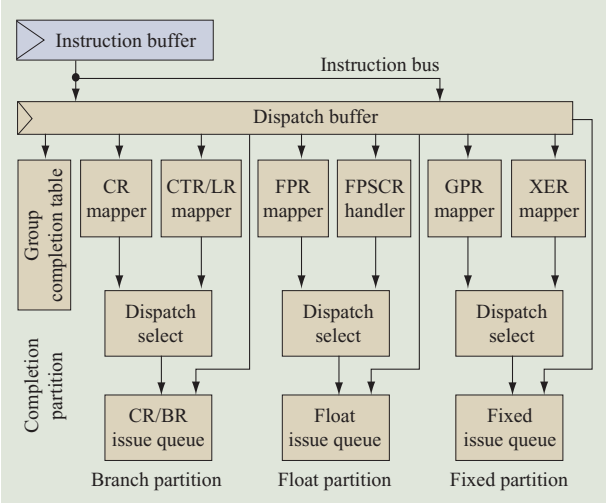

Figure 17

Major partitions of the ISU (branch, float, fixed, and completion).

setup and dispatch cycles, much computation is performed to determine whether or not the dispatch attempt will have to be rejected. This mechanism is in place so that dispatch is not arbitrarily delayed until we know for sure that it is safe. All conditions for rejecting dispatch are ORed together to yield a possible dispatch_reject in the cycle following dispatch_valid. At this point, the ISQs, mappers, completion table, and so on must all recover to their state previous to the dispatch attempt. Conditions for rejecting a dispatch attempt include the following:

- The GPR, FPR, XER, CR, or CTR/LR mapper is full.
- The fixed-point, floating-point, conditional register (CR), or branch register (BR) issue queue is full.
- The sync instruction is waiting for load.
- The scoreboard checking iop is waiting for the scoreboard setting iop.
- The exception is processed by the completion unit.
- The load reorder queue (LRQ) or store reorder queue (SRQ) is full.
- The branch instruction queue (BIQ) is full or btag is not yet written.
- Dispatch throttling occurs.
- Debug workarounds occur.

The early allocation signals, such as early_disp_i0_alloc in Figure 13, are L1-launched. They are asserted per instruction per destination field a half-cycle before dispatch_valid, if that instruction slot has a valid instruction and will write a result to that destination.

The early arch signals, disp_arch_i<0:3> in Figure 7, are L1-launched. They are asserted per instruction per destination field a half-cycle before dispatch_valid, if that allocated destination is the youngest allocation to a given lreg. The early repl signals, disp_repl_i<0:2> in Figure 8, are L1-launched. They are asserted per instruction per destination field a half-cycle before dispatch_valid, if, for that allocated destination, there is a younger instruction allocating the same lreg. As an example, consider the dispatch group for some destination, say gpr d0:

```
i0 writes lreg 5
i1 writes lreg 3
i2 writes lreg 5
i3 not valid
```

For such an instruction group, one has

```
i0: alloc=1, arch=0, repl=1
i1: alloc=1, arch=1, repl=0
i2: alloc=1, arch=1, repl=0
i3: alloc=0, arch=0, repl=0
```

Here, the i0 destination is replaced and not left in the architected state, since a younger instruction, i2, wrote to the same lreg (lreg 5).

The dispatch select logic gathers source rtags from local mappers and destination rtags from local and remote mappers. It then selects appropriate rtags on the basis of intergroup dependences and register types. It is primarily a post-mapper rtag mux to accommodate same-cycle and previous-cycle lreg dependences. Because of the latency to update the mapper state upon dispatch of a newly allocated destination, dependent sources in the same dispatch group or next dispatch group cannot use the physical rtag mapping presented by the mapper. So, in the worst case, an instruction in slot i3 that has a particular lreg as a source must choose between the mapper source rtag output and the allocated rtags for instructions i0, i1, or i2, or the last-cycle-allocated rtags for i0, i1, i2, or i3. This selection is based on source-to-destination lreg comparisons done in the cycle before dispatch. The dispatch select logic also selects appropriate load and store tags for each load or store instruction dispatched. The dispatch select is distributed as shown in Figure 17 to feed the ISU partitions.

Rename resources in the IBM POWER4 chip

The ISU contains five different mappers for managing five separate register files. All mappers are identical in functions, but their dimensions vary according to their specific requirements. Table 1 lists the rename resources implemented in the POWER4 chip.

The register mappers described in the present paper constitute the rename resources used in the IBM POWER4 chip that provides the processing power for the eServer p690. The p690 is the recently introduced high-end IBM 64-bit POWER4-architecture, 8- to 32-way server system [10, 11]. The POWER4 chips were fabricated in the IBM 0.180-μm CMOS 8S3/SOI (silicon-on-insulator) technology with seven levels of copper wiring. Features of the technology and the characteristics of the POWER4 chip are described in [11]. The chip has been operated at clock frequencies exceeding 1.3 GHz. The circuit and physical methodology used in the POWER4 chip is also described in [11].

Summary and conclusion

We applied a new nonconventional approach to the design and organization of renaming register mappers. An optimal CAM configuration was developed for designing area-efficient CAM circuit topology and for organizing the register mapper. Such a structure allows the implementation of the register mapper as a group of custom arrays. Each array is dedicated to execute a specific function. The entire mapper is partitioned into eight special custom arrays to execute the various functions of the register mapper.

We made a novel implementation of the register mapper to provide rename resources for the IBM POWER4 chip. Such rename resources facilitate the application of out-of-order and speculative executions in the processor. That, in turn, allowed for a high level of concurrency in the pipeline and made a substantial contribution to superior machine performance.

Acknowledgments

The authors wish to acknowledge our many colleagues on the POWER4 design team. We would especially like to thank Sam Chu, Peter Klim, Joel Silberman, and Nathan Peterson for their discussions and interests, and Ray East for his support in program management.

References

1. R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Res. & Dev.* **11,** No. 1, 25–33 (January 1967).
2. J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors," *Proc. IEEE* **83,** No. 12, 1609–1624 (December 1995).
3. A. Moshovos and G. S. Sohi, "Microarchitectural Innovations: Boosting Microprocessor Performance Beyond Semiconductor Technology Scaling," *Proc. IEEE* **89,** No. 11, 1560–1575 (November 2001).
4. R. E. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro* **19,** No. 2, 24–36 (March/April 1999).

5. R. E. Kessler, E. J. McLellan, and D. A. Webb, "The Alpha 21264 Microprocessor Architecture," *Proceedings of the International Conference on Computer Design (ICCD '98)*, October 1998, pp. 90–95.
6. J. Leenstra, J. Pille, A. Muller, W. M. Sauer, R. Sautter, and D. F. Wendel, "A 1.8-GHz Instruction Window Buffer for an Out-of-Order Microprocessor Core," *IEEE J. Solid-State Circuits* **36,** No. 11, 1628–1635 (November 2001).
7. D. S. Henry, B. C. Kuszmaul, G. H. Loh, and R. Sami, "Circuits for Wide-Window Superscalar Processors," *Proceedings of the 27th International Symposium on Computer Architecture,* June 2000, pp. 236–247.
8. K. J. Schultz, "Content-Addressable Memory Core Cells: A Survey," *Integration, the VLSI Journal* **23,** No. 2, 171–188 (November 1997).
9. S. Jones, "Design, Selection and Implementation of Content-Addressable Memory for a VLSI CMOS Chip Architecture," *Computer and Digital Techniques, IEE Proc.* **135,** No. 3, 165–172 (May 1988).
10. J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy, "POWER4 System Microarchitecture," *IBM J. Res. & Dev.* **46,** No. 1, 5–25 (January 2002).
11. J. D. Warnock, J. M. Keaty, J. Petrovick, J. G. Clabes, C. J. Kircher, B. L. Krauter, P. J. Restle, B. A. Zoric, and C. J. Anderson, "The Circuit and Physical Design of the POWER4 Microprocessor," *IBM J. Res. & Dev.* **46,** No. 1, 27–51 (January 2002).

Received October 30, 2003; accepted for publication February 9, 2004; Internet publication November 24, 2004

Taqi N. Buti *IBM Systems and Technology Group, 2455 South Road, Poughkeepsie, New York 12601 (buti@us.ibm.com).* Dr. Buti is a Senior Engineer in the POWER6 development team leading the circuit design of instruction dispatch. He led the circuit design of the register rename mappers for the POWER4 and POWER5 processors. He has worked on the cache, cache TAGS, TLB, BAT, SR, and register files for the 603ev, 604, and 620 PowerPC processors. He also designed a flash EEPROM array and charge pump. He received a Ph.D. degree in physics from the Massachusetts Institute of Technology. His early work was in the field of medium- and high-energy physics, investigating and synthesizing electro-excitation and hadronic interaction experiments. After his postdoctoral work at MIT, Dr. Buti joined Harris Semiconductor in 1984 to work on a variety of problems related to semiconductor device physics and technology. He joined IBM in 1988 at the East Fishkill Semiconductor Laboratory, where he was engaged in CMOS, SOI, and BiCMOS device and technology, and device and process design, simulation, and characterization. He co-invented the halo source GOLD drain asymmetrical FET. Dr. Buti has published more than 30 papers and holds 14 patents.

Robert G. McDonald *University of Texas at Austin, 1 University Station C0500, Austin, Texas 78712 (robertmc@cs.utexas.edu).* While at IBM, Mr. McDonald helped develop several high-performance processors, including the POWER2 and POWER4 processors. He contributed to the definition, modeling, and performance tuning of the overall POWER4 core microarchitecture, and also served as the primary architect and logic designer for the original POWER4 register mappers. Mr. McDonald studied electrical engineering at Texas A&M University and the Massachusetts Institute of Technology. He has filed more than 24 patent applications.

Zakaria Khwaja *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (khwaja@us.ibm.com).* Mr. Khwaja joined IBM in 1997 in the POWER4 design team as a circuit designer. He worked in POWER4 and POWER5 processors in the development of the floating-point register file, instruction sequence unit mapper, and issue queue circuits. Currently he leads a circuit design team for POWER5 and its follow-on designs. Prior to joining IBM, he worked at Advanced Micro Devices as a designer in the microprocessor and the chipset groups. Mr. Khwaja received an M.S. degree in electrical engineering from Louisiana State University, where he did research on heterojunction semiconductor devices. His technical interests are in the areas of array design, low-power design, and design methodology.

Asit Ambekar *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (ambekar@us.ibm.com).* Mr. Ambekar joined IBM in 1997 in the POWER4 design team as a circuit designer. He received an M.S. degree in electrical engineering from Texas A&M University. He worked in POWER4 and POWER5 processors in the development of the floating-point status control register file, instruction sequence unit mapper, and dispatch and completion unit circuits. Mr. Ambekar has been an integral part of the array design team for POWER processors since joining IBM. His current responsibilities include circuit design for the POWER5 processor and follow-on designs, and POWER6 circuit design for the instruction decode unit and register file design.

Hung Q. Le *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (hung@us.ibm.com)*. Mr. Le is a Distinguished Engineer in the POWER6 development team. He joined IBM in 1979 after graduating from Clarkson University with a B.S. degree in electrical and computer engineering. He has worked on the development of several mainframe products. Since 1991, he has worked on the development of the PowerPC microprocessor, and POWER3, POWER4, and POWER5 products. His technical interests are in the field of processor design involving superscalar, out-of-order, and multithreading design. Mr. Le received an IBM Corporate Award and two IBM Outstanding Technical Awards for his work on mainframe and POWER processor development. He holds 48 patents.

William E. Burky *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (burky@us.ibm.com)*. Mr. Burky is a Senior Engineer in the POWER5 development team. He joined IBM in 1991 after receiving a B.S. degree in computer engineering from Carnegie Mellon University. He has since earned a M.S.E.E. degree from National Technological University. He has worked on the development of the PowerPC system ASICs and POWER3, POWER4, and POWER5 microprocessors. He currently leads the POWER5 instruction sequencing unit design team, specializing in multithreading design, instruction dispatch, and exception handling. Mr. Burky holds four patents, with 14 patents pending; he has received a Fifth Plateau IBM Invention Achievement Award.

Bert Williams *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (bertw@us.ibm.com)*. Mr. Williams joined IBM in 1984 after receiving a B.S. degree in electrical engineering from the University of Texas. He has worked on the POWER2, POWER3, POWER4, and POWER5 microprocessor designs, and is currently working on the POWER6 data prefetch engine.