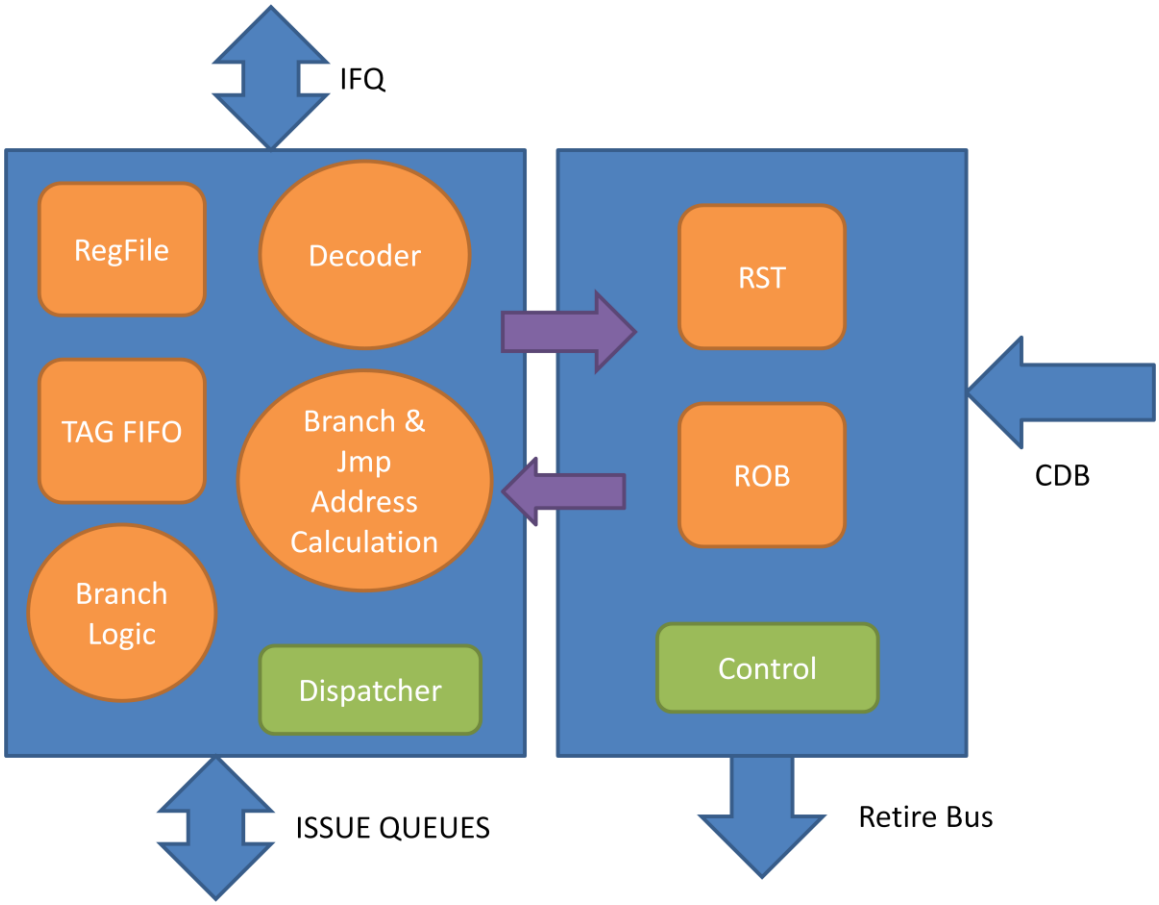
1 Introducción

La unidad de despacho es responsable de leer instrucciones del IFQ y enviarlas a las respectivas colas de ejecución. Dentro de la unidad de despacho las instrucciones son procesadas una a la vez y en orden de programa. La unidad de despacho es responsable de las siguientes tareas.

- Leer la instrucción de la IFQ.
- Decodificar las instrucciones.
- "Despachar" la instrucciones a la correspondiente cola de ejecución.
- Actualizar la tabla de estado de los registros (RST) y el banco de registros.
- Calcula las direcciones de brinco para las instrucciones *Jump* y *Branch.*
- Ejecuta las instrucciones *Jump*.
- Reservar una localidad en el Re-order Buffer



La Unidad de Despacho interactúa con la IFQ, con las cuatro colas de ejecución (Issue Queues) y también con el bus de datos común (CDB).

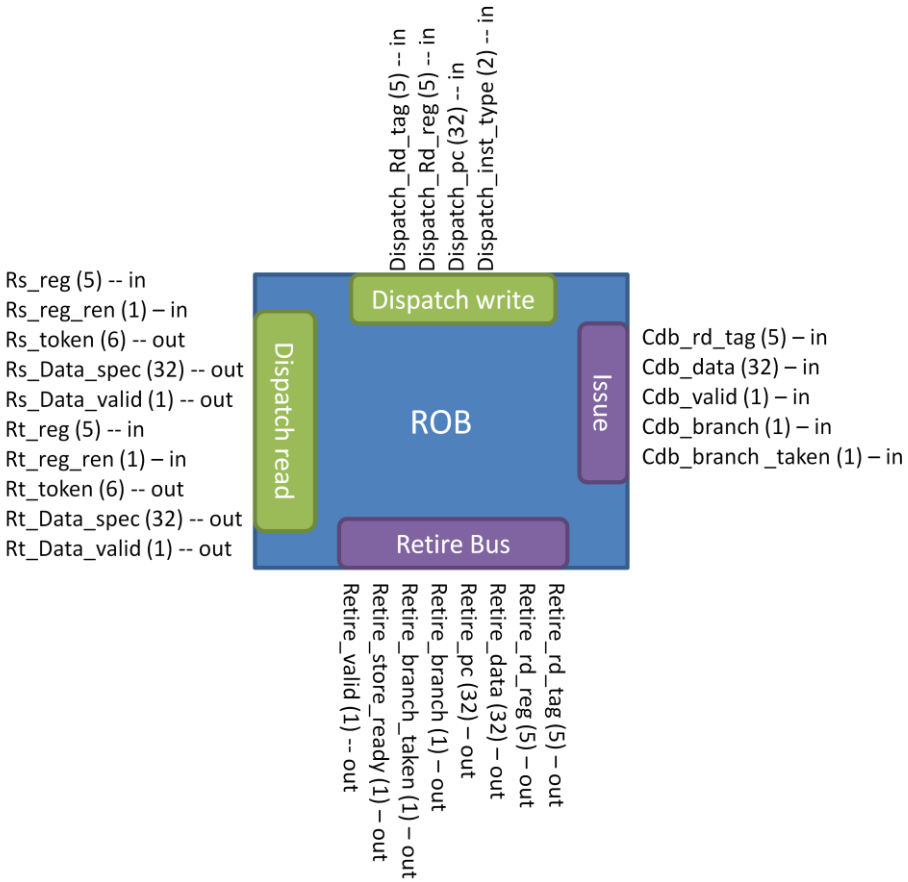
El banco de registros, el RST y el TAG FIFO se encargar de implementar el mecanismos de re-nombramiento de registros (REGISTER RENAMING). La unidad de decodificación funciona igual que en la implementación del pipeline de 5 estados. Para todas las instrucciones, sean o no *branches* o *jumps*, se calculan ambas direcciones de salto, la del *branch* y la del *jump*. Finalmente el despachador se encarga de ensamblar las señales transmitidas a las colas de ejecución.
La unidad de despacho debe de ser capaz de hacer el re-nombramiento de registros, decodificación de la instrucción y mandar la instrucción a su respectiva cola de ejecución en dos ciclos de reloj. En un ciclo de reloj se decodifica la instrucción y se inicia la consulta del estado de los operandos de dicha instrucción, en el segundo ciclo se ensamblan las señales que tienen que ser transmitidas a las diferentes colas de ejecución y se actualizan las respectivas entras del RST y del ROB.

- Se leen los registros Rs y Rt tanto del banco de registros como de la RST-ROB, se asume que todas las operaciones requieren de estos operandos.
- Se calculan las direcciones de salto tanto para el caso del branch como del jump. Se asume que todas las instrucciones son branches o jumps.
- A los 16 bits del campo inmediato se les hace la extensión de signo, se asume que todas las instrucciones son del tipo I.
- Se decodifica la instrucción.
- Dependiendo del tipo de instrucción (resultado de la decodificación), del valor de los estados de los registros leídos (resultado RST-ROB) y de los resultados publicados en el CDB-Retire Bus (si alguno) se generan y seleccionan las señales que deben de ser propagadas a las colas de ejecución.
- En caso que la instrucción no pueda ser despachada (la cola de ejecución destino se encuentre llena) se tienen que retener los resultados de la decodificación.
- En caso que la instrucción si pueda ser despachada se disparan las siguientes operaciones. Actualización del RST-ROB y mandar a traer la próxima instrucción.
- Cada vez que el CDB-Retire Bus publique un nuevo resultado se disparan las siguientes operaciones, actualización del RST-ROB y del banco de registros.

2 Especificaciones de Diseño.

2.1 Re-Order Buffer

El Re-Order Buffer es necesario para poder sopar el modelo de “precise exceptions” y tambien para poder tener ejecucion especulativa. En general el ROB se encarga de que todas las instrucciones sean “retiradas” en el orden del programa. Hasta el momento cada vez que el dispatch se encuentra con una instrucción del tipo branch se tiene que detener el pipeline hasta que la condición de brinco sea resuelto. Una vez que el ROB es implementado, podemos incluir un motor de predicción de saltos tan simple como “always taken” o “always non-taken” para seguir ejecutando código especulativamente y si la predicción es correcta salvamos ciclos de reloj. En caso contrario el ROB necesita tener un mecanismo para revertir/ignorar las instrucciones que se ejecutaron especulativamente.



EL ROB se comunica con el Dispatch, con el CDB y es dueño del “Retire Bus”.
Desde el Dispatch recibe dos tipos de peticiones/operaciones. 1) consulta el status de los registro fuente (pendiente en el back-end o especulativamente en el ROB). 2) escribir una nueva entrada en el ROB.

Por el CDB actualiza las entradas del ROB y el Retire Bus actualiza el banco de registros, retira las instrucciones del tipo store e inicia el mecanismo de roll-back.

2.1.2 Interfaces.

Dispatch consulta estatus de los registros fuente.

Rs_reg (5) – in
Rs_reg_ren (1) – in
Rs_token(6) -- out
Rs_Data_spec (32) -- out
Rs_Data_valid (1) – out // si Rs_Data_valid es ‘1’ especulativa en el ROB
Rt_reg (5) -- in
Rt_reg_ren (1) – in
Rt_token (6) -- out
Rt_Data_spec (32) -- out
Rt_Data_valid (1) – out // si Rt_Data_valid es ‘1’ especulativa en el ROB

Dispatch escribe nueva entrada en el ROB

Dispatch_Rd_tag (5) – in // el tag asignado por el TAG FIFO, ahora TODAS las instrucciones llevan un TAG.
Dispatch_Rd_reg (5) – in
Dispatch_pc (32) – in // PC de la instruccion a la cual corresponde Rd_tag, en el caso de los branches es la direccion de salto.
Dispatch_inst_type (2) – in
- 00 – instruccion donde el rd_tag es válido.
- 01 – branches
- 10 – stores.

CDB

Cdb_tag: TAG de 5 bits
Cdb_valid: senial que indica si el TAG es válido.
Cdb_data: 32 bits del valor que debe de tomar el registro asociado con el TAG.
Cdb_branch: Indica si la instrucción que termino ejecución se trata de un branch.
Cdb_branch_taken: ‘1’ indica que el branch debe de ser tomado, ‘0’ no debe de ser tomado.

Retire Bus

Retire_rd_tag (5) – out
Retire_rd_reg (5) – out // registro que tiene ser actualizado en el banco de registros
Retire_data (32) – out
Retire_pc (32) – out // solo tiene sentido en caso de un branch mal predicho.
Retire_branch (1) – out // si se trata de un branch la instruccion que fue retirada

Retire_branch_taken (1) – out // se el branch tiene que ser tomado (prediccion equivocada)
Retire_store_ready (1) – out // si se trata de un store la instruccion que fue retirada
Retire_valid (1) – out //indica si se esta retirando una instruccion.

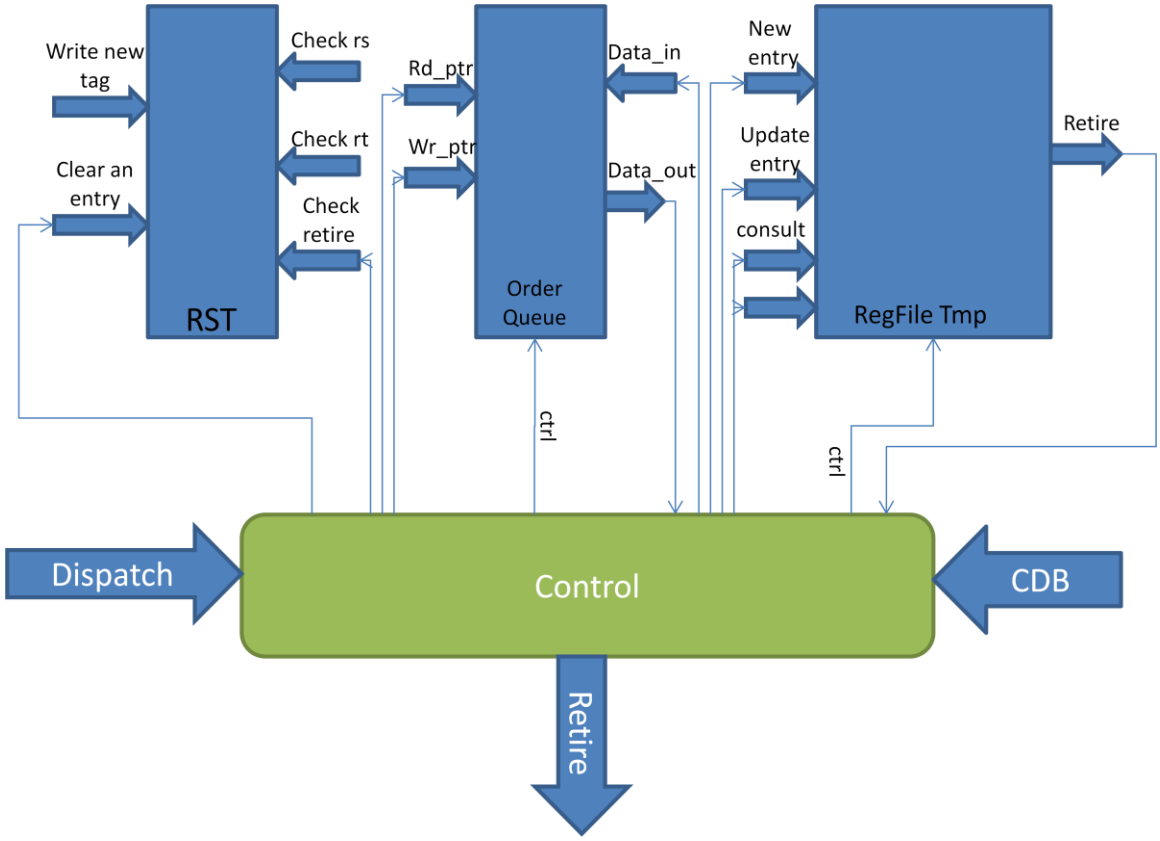
2.1.3 Renombramiento de los Registros.

Entre el banco de registros, el RST (register status table) y el TAG FIFO se lleva a cabo el mecanismo de renombramiento de registros.

Para cada instrucción con campo Rd se le asigna un TAG (o el alias del registro destino) como mínimo se necesitan tantos TAGs como registros existan, en esta implementación vamos a tener el mínimo requerido (32 TAGs). Como “00000” es un TAG valido necesitamos un bit que indique la valides del TAG. A la combinación TAG + bit valido le vamos a llamar TOKEN (6 bits).

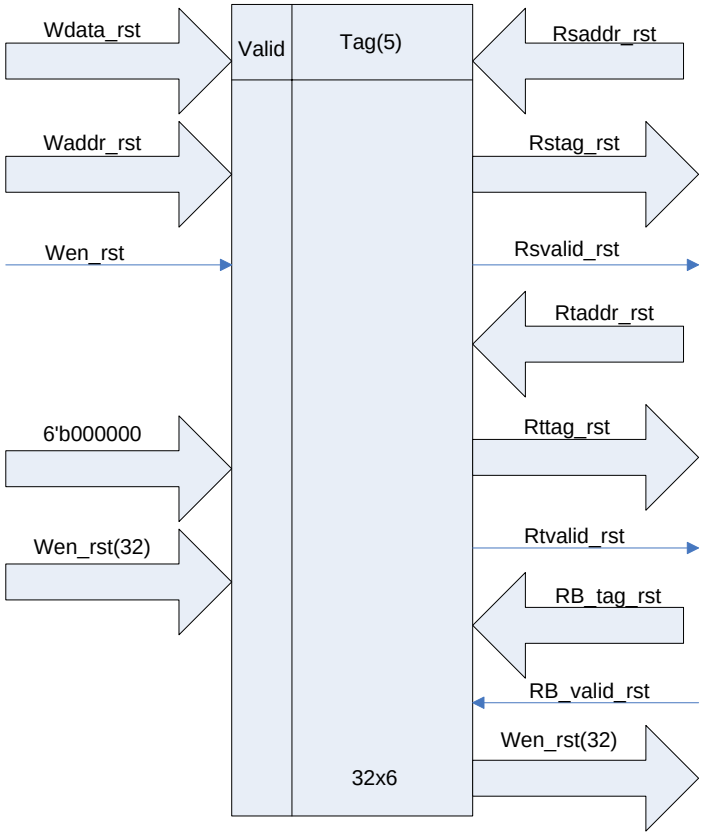
El operando de una instrucción se puede encontrar en tres estados. Valido en Banco de Registro, Especulativo en el ROB, o pendiente en el BackEnd. Para poder inferir en qué estado se encuentran los operandos de una instrucción es necesario consultar simultáneamente el banco de registros el RST y el ROB.

La siguiente figura muestra el diagrama de la estructura encargar de mantener el orden del programa y el status de los registros físicos.

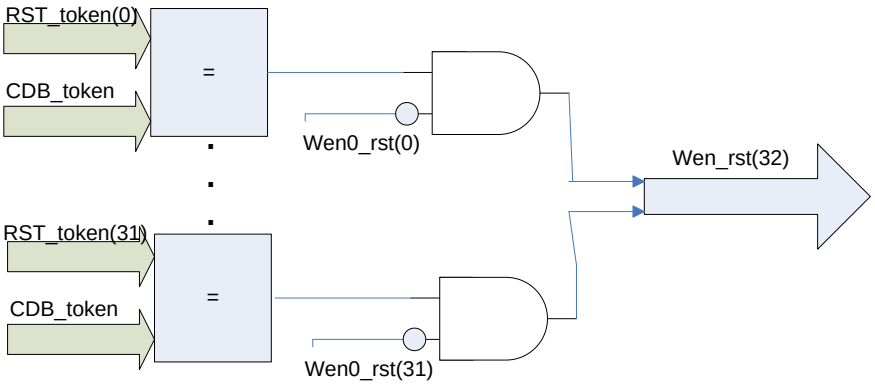


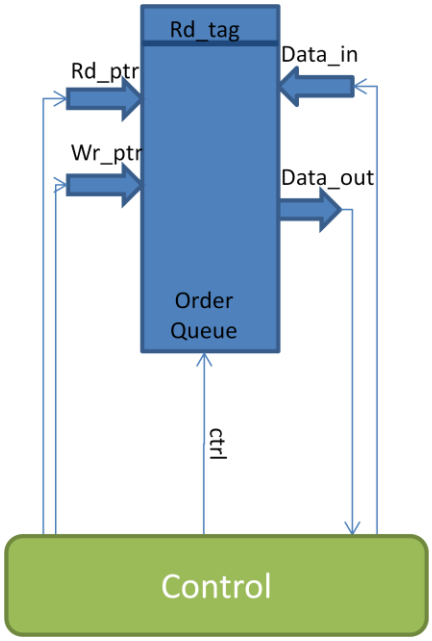
Se necesitan dos puertos de escritura y dos puertos de lectura en el RST. Un puerto de escritura para escribir el TAG de la instrucción siendo despachada y el segundo puerto de escritura para limpiar el TAG publicado en el CDB. Uno de los puertos de lecturas es para el registro rs y el otro para el registro rt de la instrucción siendo despachada.

Cuando un TAG es publicado en Retire Bus (liberado) todos los tags del RST (con bit valido activo) son comparadas contra el tag del Retire Bus (con bit valido activo). La entrada donde la comparación haga match es limpiada y el banco de registros es actualizado. Es por esto que se utilizan 32 bits 1-hot enconded para la señal Wen_rst(32).



Puede suceder que en el mismo ciclo se quiera limpiar y escribir un nuevo TAG en la misma localidad del RST, para este caso se tiene que dar prioridad a la escritura del nuevo TAG.





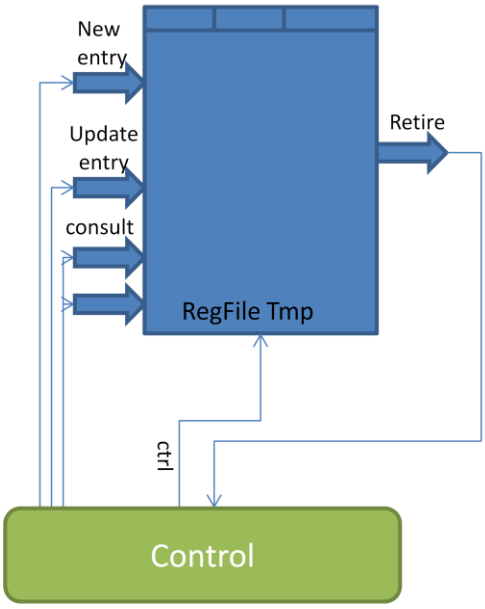
El Order Queue es de 32 entradas, y se comporta como una FIFO circular manejada por un apuntador de lectura y uno de escritura.

En este cola se lleva el orden en el que las instrucciones son despechadas y los Rd tags son asignados.
Data_in = Rd_tag.
Data_out = Rd_tag apuntado por el Rd_ptr, siempre presente.

El controlador del ROB se encarga de controlar el rd_ptr, el wr_ptr y de generar la señal de write enable.

Cada vez que por el canal del Dispatch llega un nuevo Rd_tag este se escribe en la localidad que apunta el wr_ptr y se incrementa el mismo.

Cada vez que la instrucción que esta al tope de la Order Queue puede ser retirada se incrementa el Rd_ptr.



Rd_reg	Pc	Inst_type	Spec_Data	Spec_valid	Valid

El RegFile Tmp es de 32 entradas y se direcciona con el valor del Tag ya sea Rd_tag para nuevas y actualizaciones o rs_tag y rt_tag para consultas.

New_entry: cada vez que llega un nuevo Rd_tag por el canal de dispatch se guarda una nueva entrada en el RegFile tmp. Los campos que se inicializan son los siguientes. {Rd_reg, Pc, inst_type, Valid}

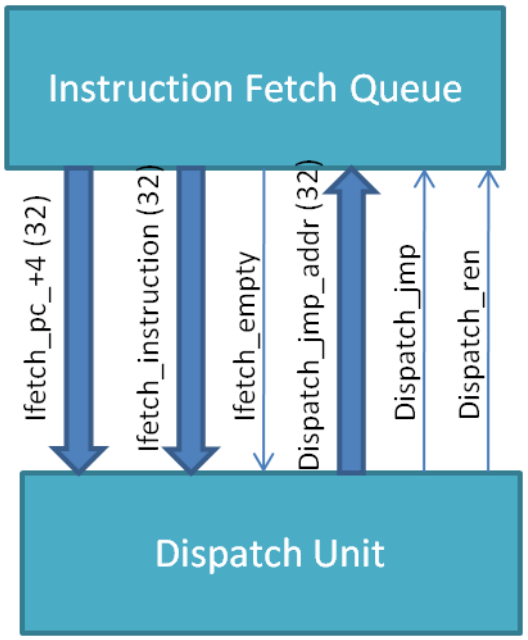
Update entry: cada vez que se publica un nuevo dato en el CDB se actualizan los campos de Spec_Data y Spec_valid.

Consult: cada vez que se consulta el estado de los registro fuente rs y rt, también se consulta si existe el data guardado especulativamente en el RegFile tmp.

Cada vez que la entrada a al que apunta el rd_ptr del order queue se encuentra completada (spec_valid y valid) esta puede ser retirada del ROB.

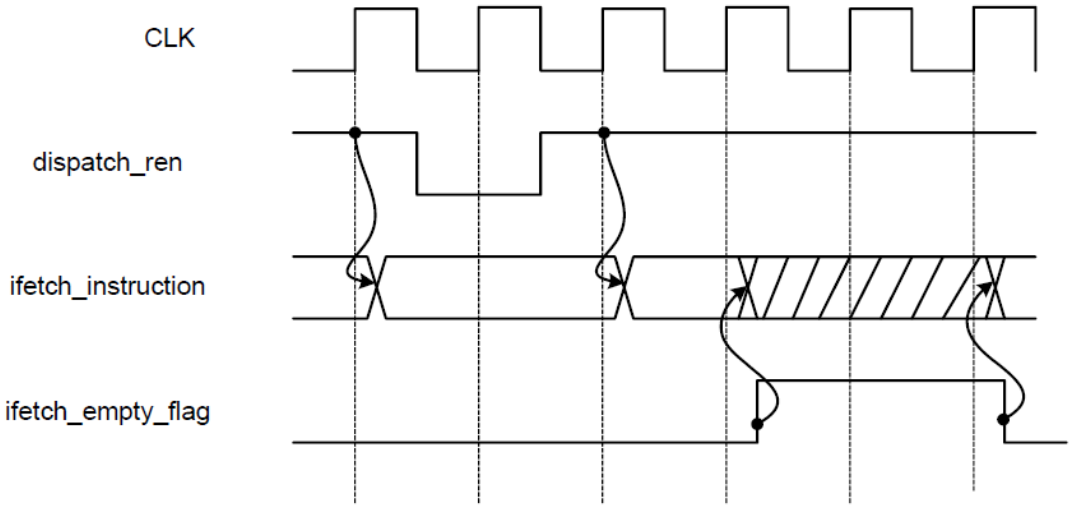
2.2 Dispatcher.

2.2.1 Lectura de Instrucciones.

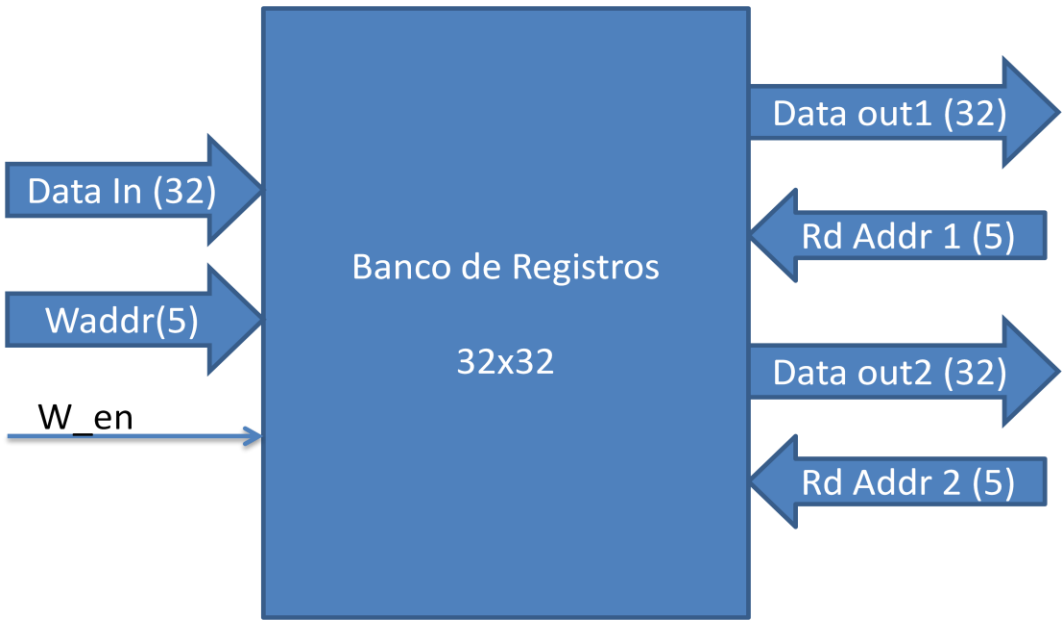


- ifetch_instruction: 32 bits de la instrucción.
- ifetch_empty_flag : la instrucción es valida is '0', invalida si '1'.
- ifetch_pc_plus_four: valor de PC+4 para el cálculo de las direcciones de salto.
- dispatch_ren: si '1' el IFQ incrementa el apuntador de lectura y muestra una nueva instruccion, si '0' el IFQ sigue mostrando la misma instrucción.
- dispatch_jmp_br: '1' la instrucción es un jump o un branch que fue tomado.
- dispatch_jmp_br_addr: 32 bit dirección de salto.

A continuación se muestra un diagrama de tiempo de la interacción entre la IFQ y la unidad de despacho.



2.2.2 Banco de Registros.

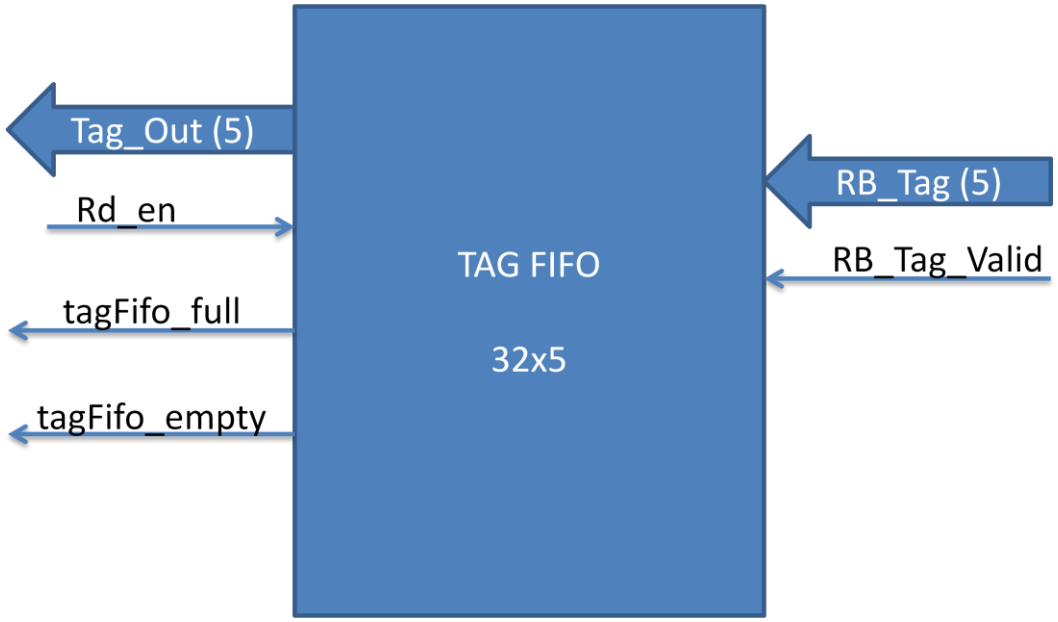


2.2.2 TAG FIFO.

Los TAGs son generados por el TAG FIFO, la cual es una simple FIFO de tamanio 5 x 32. Tiene un apuntador de escritura y lectura de 6 bits (rp y wp respectivamente). WP es inicializado con el valor de “100000” y el RP es

inicializado a “000000”. Las entradas del 0 al 31 son inicializadas con los números del 0 al 31. Entonces la FIFO inicialmente se encuentra llena.

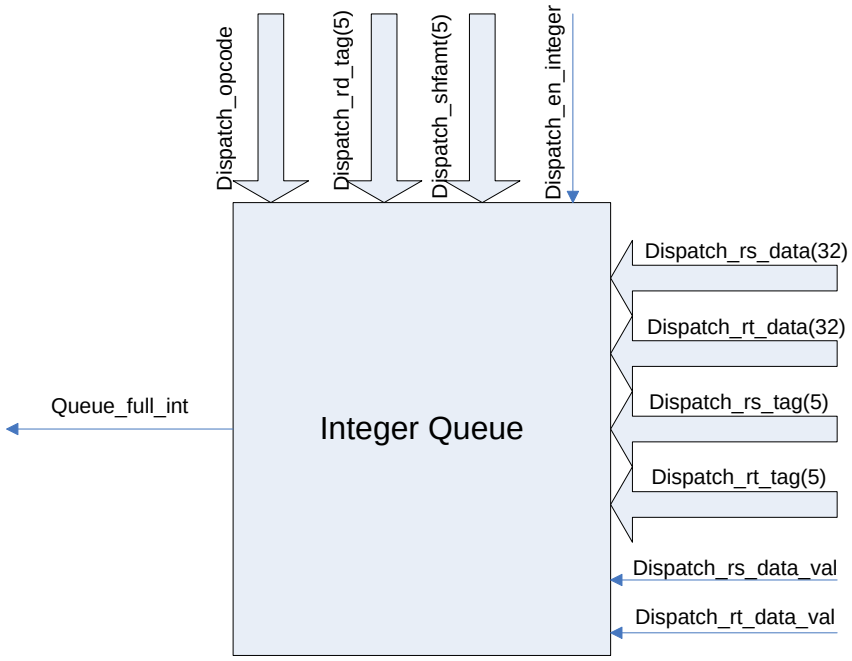
Cuando una instrucción es despachada, el apuntador de lectura es incrementado y el siguiente TAG es leído. Cuando el Retire BUS publica una TAG valido, el TAG es escrito y el WP es incrementado. La unidad de despacho solo levanta la senial de lectura cuando una instrucción con registro destino puede ser despachada. La unidad de despacho no lee un TAG cuando la instrucción no tiene un registro destino o no puede ser despachada.

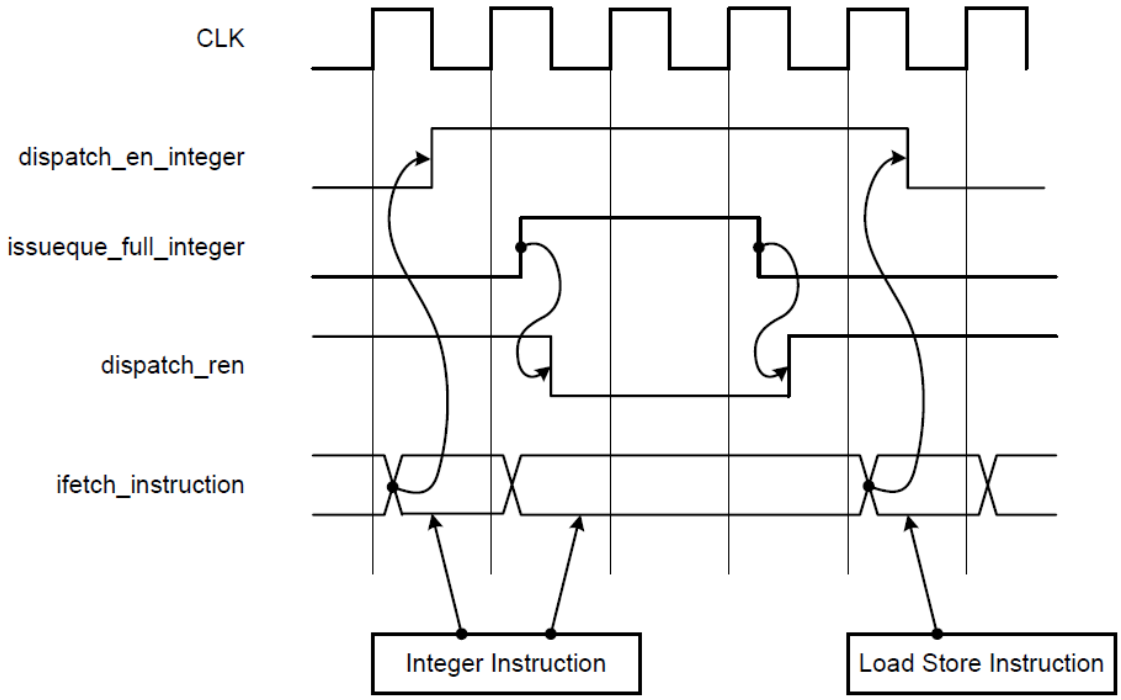


2.2.3 Interface con las colas de ejecución.

Cuando una instrucción ha sido decodificada y los otros campos están listos, la señal que habilita la correspondiente cola de ejecución es habilitada para indicar la intención de la unidad de despacho de escribir en ella. Si la cola de ejecución destino se encuentra llena activa la bandera “llena” para indicarle a la unidad de despacho que no puede recibir la instrucción. Como consecuencia la unidad de despacho detiene la IFQ, deshabilita la lectura de la TAG FIFO y mantiene la señal que habilita la cola de ejecución activada.

Cola de Ejecucion de Enteros.





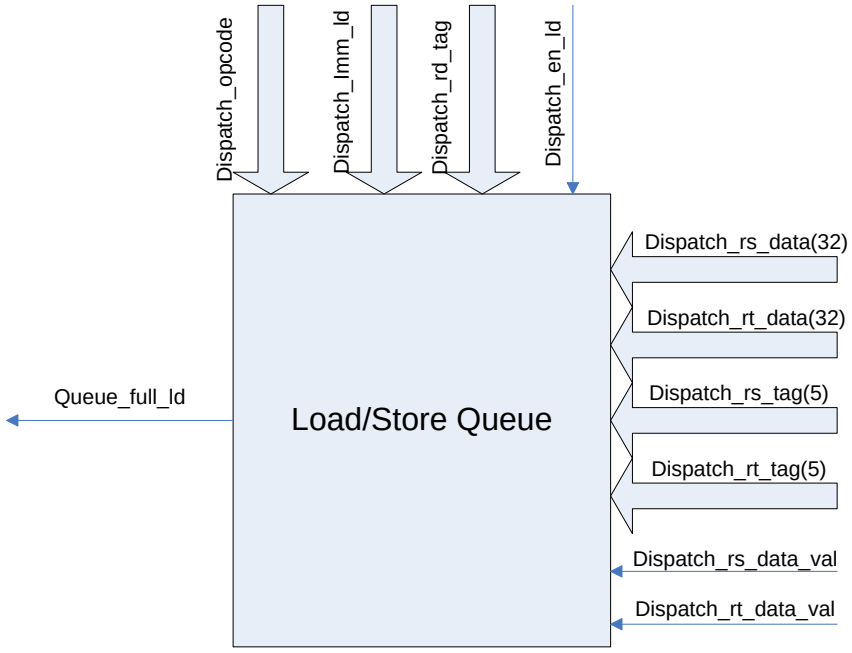
Señales comunes para todas las colas de ejecución.

dispatch_rs_data: operando rs
dispatch_rs_data_valid: es ‘1’ si rs tiene el ultimo valor, si no ‘0’.
dispatch_rs_tag: tag para rs.
dispatch_rt_data: operand rt
dispatch_rt_data_valid: es ‘1’ si rt tiene el ultmo valor, si no ‘0’.
dispatch_rt_tag: tag para rt
dispatch_rd_tag: TAG asignado al registro destino de la instrucción

Señales especificas para la cola de ejecución de enteros.

dispatch_en_inetger: unidad de despacho intenta escribir una instrucción en la cola de ejecución de enteros.
issueque_integer_full: Bandera “llena” de la cola de ejecución de enteros.
dispatch_opcode: 3-bit opcode para la ALU.
Dispatch_shfamt: 5-bits en caso de una instrucción del tipo shift

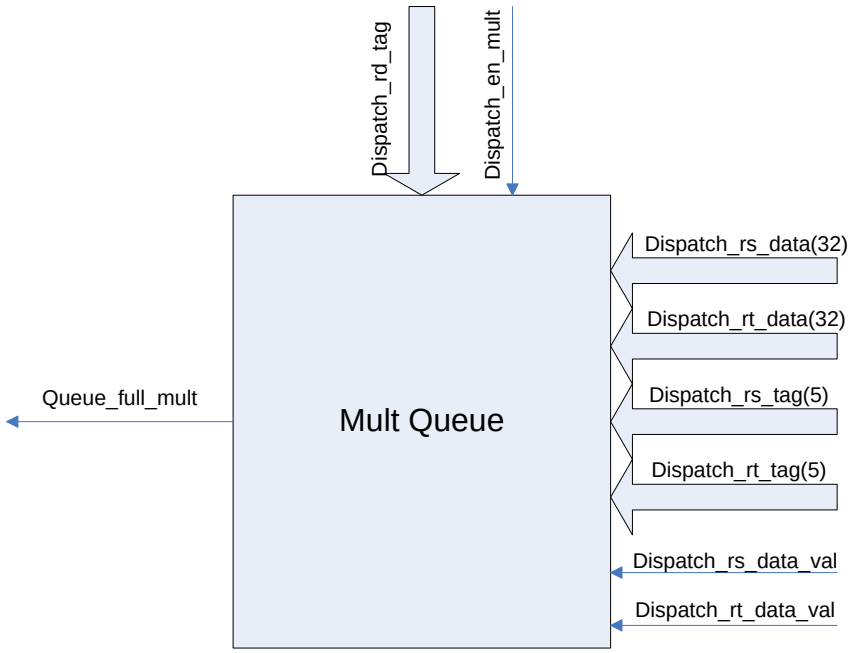
Cola de ejecución de acceso a memoria.



dispatch_en_ld_st: DU intenta escribir una instrucción en la cola LD/ST.
issueque_full_ld_st: Bandera "llena" de la cola de ejecución LD / ST.
dispatch_opcode: 1 - bit opcode para distinguir entre LD y ST.
dispatch_imm_ld_st: 16 - bit del campo inmediato para calcular la dirección de memoria.

Cola de ejecución de multiplicaciones.

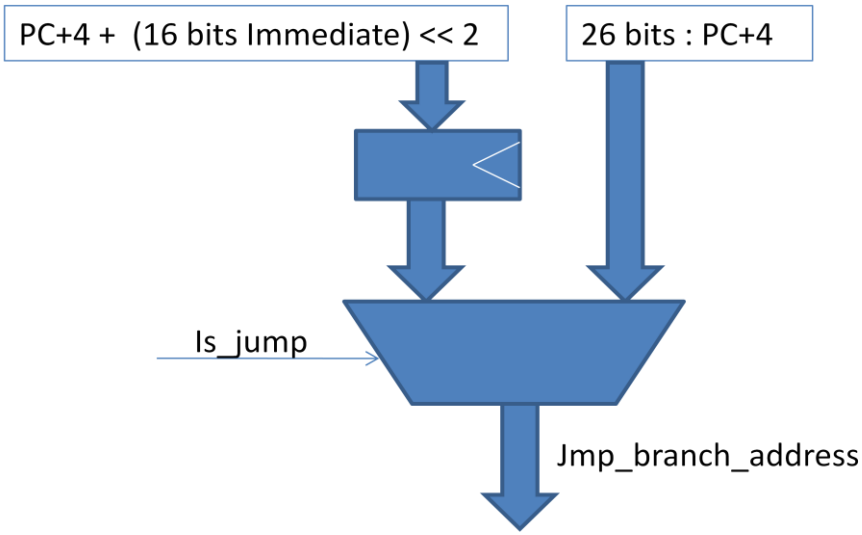
dispatch_en_mul: DU intenta escribir una instrucción en la cola Multiplicación.
issueque_mul_full: Bandera "llena" de la cola de ejecución Multiplicación.



2.2.4 Lógica para instrucciones Jump & Branch.

Las instrucciones del tipo Jump son ejecutadas por la unidad de despacho a lo igual que el cálculo de las direcciones de salto tanto para Jumps como para Branches. En caso de un Jump, la dirección es calculada combinatoriamente (lógica combinacional) y presentada al IFQ ese mismo ciclo.

La siguiente figura muestra la manera de transmitir la dirección de salto a la IFQ.


PC+4 + (16 bits Immediate) << 2
26 bits : PC+4
Is_jump
Jmp_branch_address