CS 520
Register Renaming

Timothy N. Miller

Motivating Example

```
(I1)   MUL R2, R1, R7       /* R2 <- R1 * R7 */
(I2)   ADD R4, R2, R3       /* R4 <- R2 + R3 */
(I3)   SUB R2, R6 #8        /* R2 <- R6 - 8 */
(I4)   LOAD R5, R2, #12     /* R5 <- Mem[R6 + 12] */
```

- Two active instances of R2
  - MUL ➔ R2 ➔ ADD
  - SUB ➔ R2 ➔ LOAD
- Allocate new R2 for each flow dependency

Code analogy

- x = [expression]
- y = [expression dependent on x]
- x = [another expression]
- At this point, cannot get the old x.
- These could execute in parallel

Principle of Register Renaming

- Architectural register (R2) is just a label
  - R2 is not real
- Dynamically assigned to physical register (e.g. P37) during dispatch.

Basic idea

- Dispatch for destination register (Rn)
  - Allocate free physical register (Pm)
  - Alias Rn to Pm (in alias table)
  - New instance of Rn
- Dispatch for source registers
  - Use alias table to translate Rn to Pm

- Dynamic binding of architectural registers to physical registers

```
I0:  ADD R1, R2, R3
I1:  MUL R4, R1, R6
I2:  LOAD R1, address
I3:  ADD R2, R1, R6
```

Consider now the renamed version of this instruction stream:

```
        /* Assume that Pj currently implements Rj */
  /* Assume free physical registers are P32, P33, P34 and P35 */
```

```
IO:   ADD P32, P2, P3
                   /* R1 renamed to P32 */
I1:   MUL P33, P32, P6
        /* R4 renamed to P33; note use of P32 for R1 */
I2:   LOAD P34, address
                   /* R1 renamed to P34 */
I3:   ADD P35, P34, P6
                   /* R2 renamed to P35 */
```

In this version, the delays due to anti and output dependencies are absent.

Required hardware

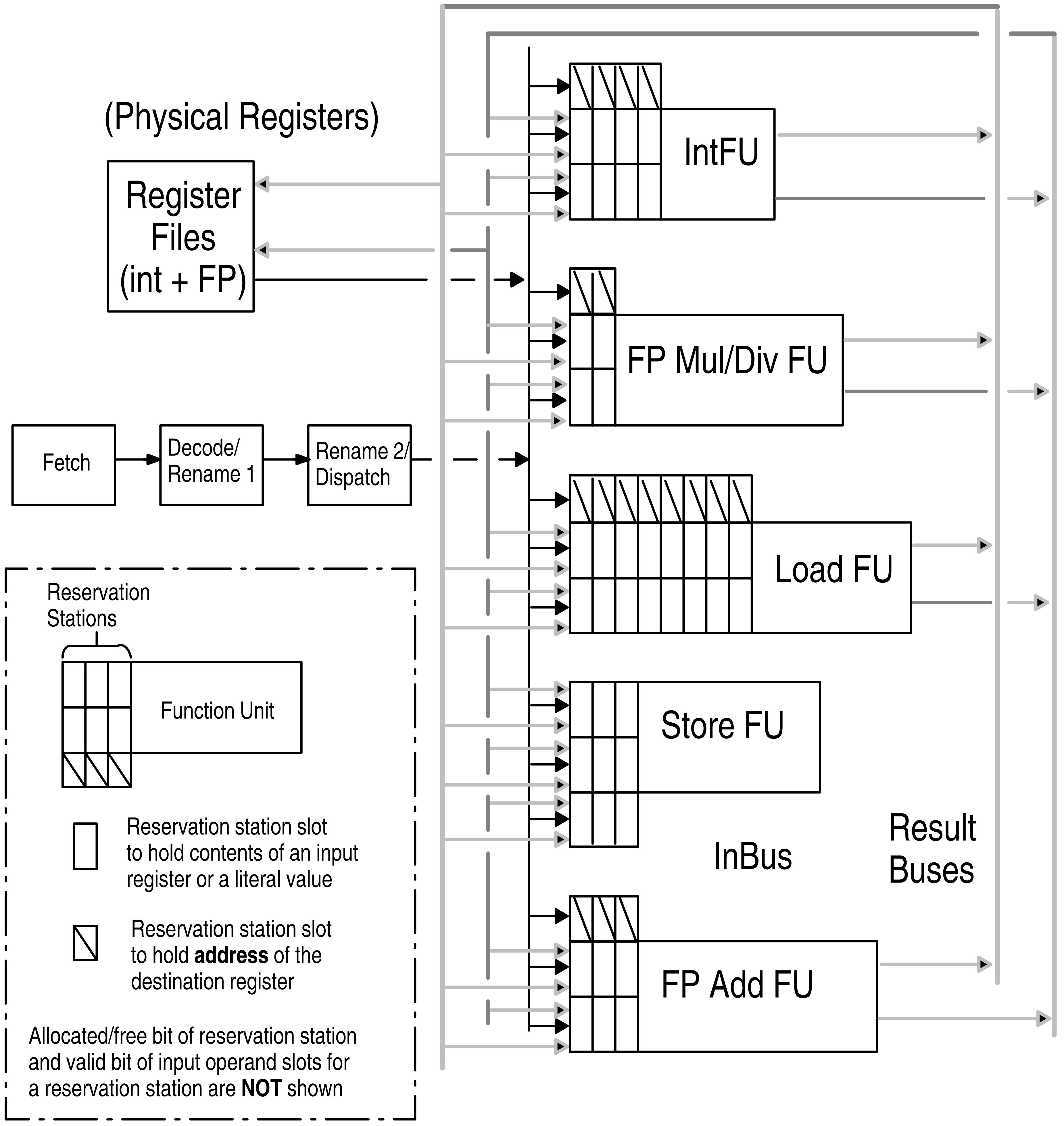
- Prevent startup before flow dependencies are satisfied.
- Manage allocation and deallocation of physical registers

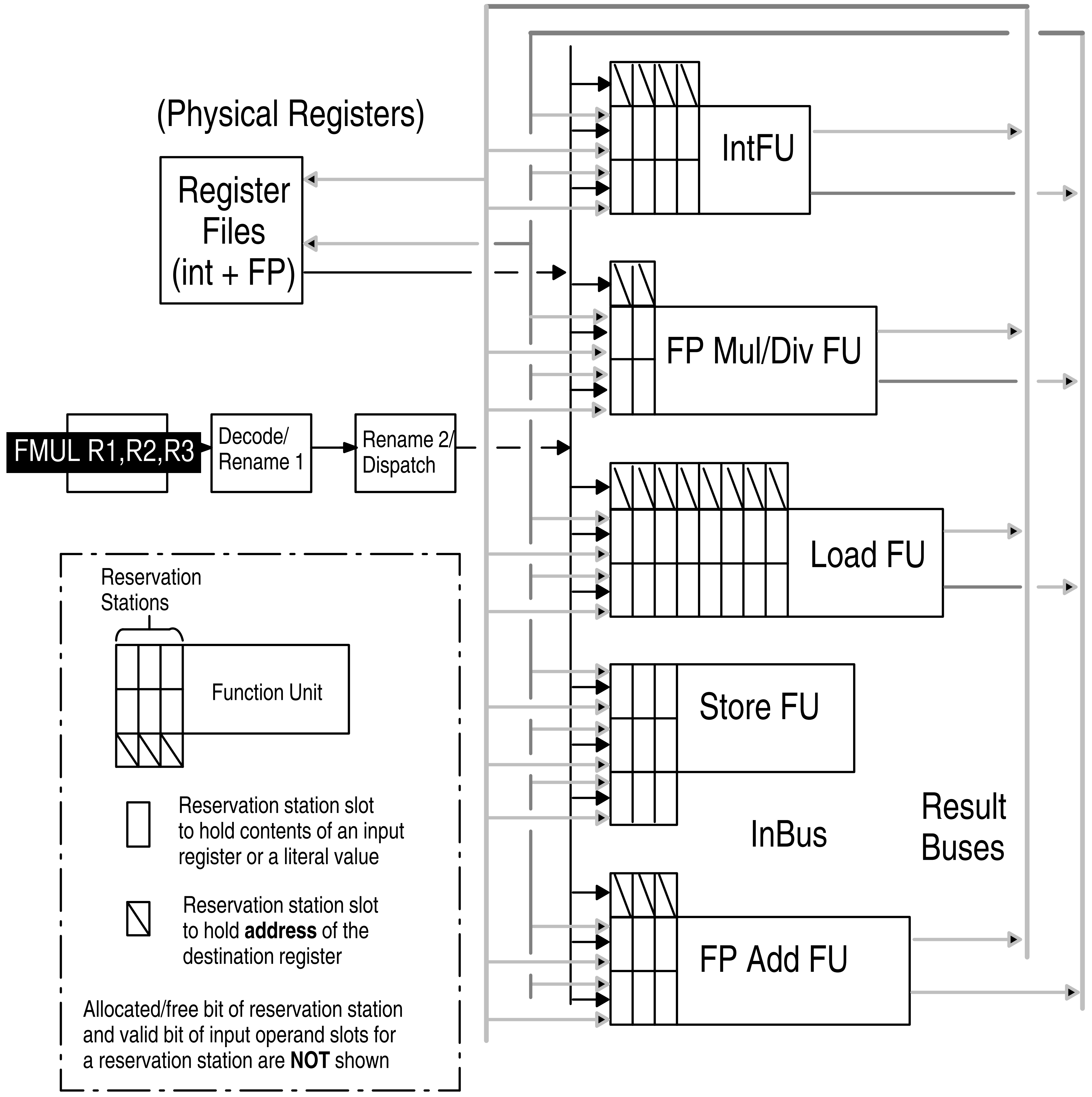
More requirements

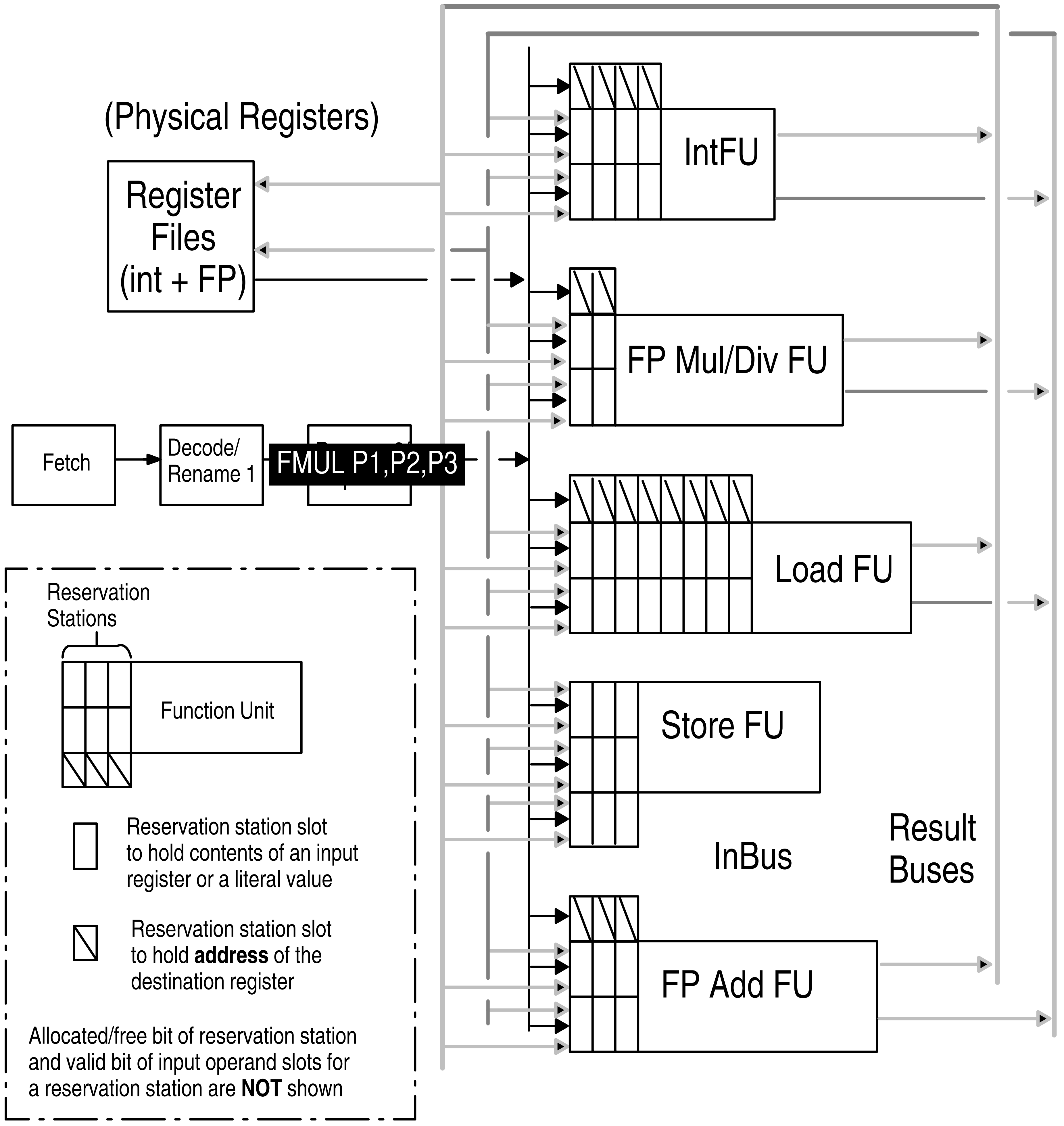
- More physical regs than architectural regs
  - Instruction window
- Mapping table from arch. to phys. reg
  - Rename Table / Register Alias Table
- Substitute phys. reg numbers for arch. reg numbers.

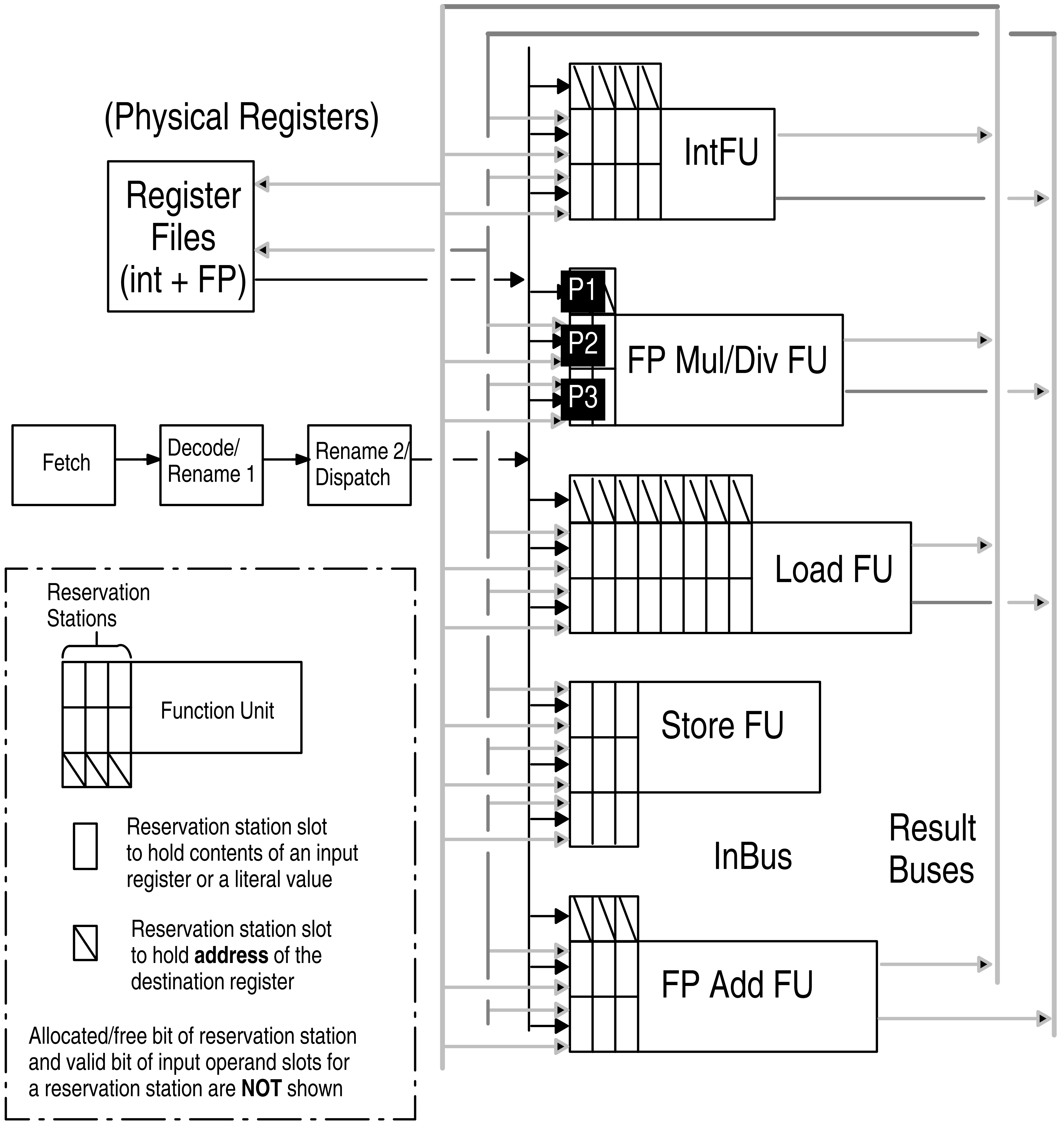
How to free a physical register

- A physical register can be reclaimed when:
  - All dependent instructions that require it as input have read it, and
  - The corresponding architectural register has been renamed

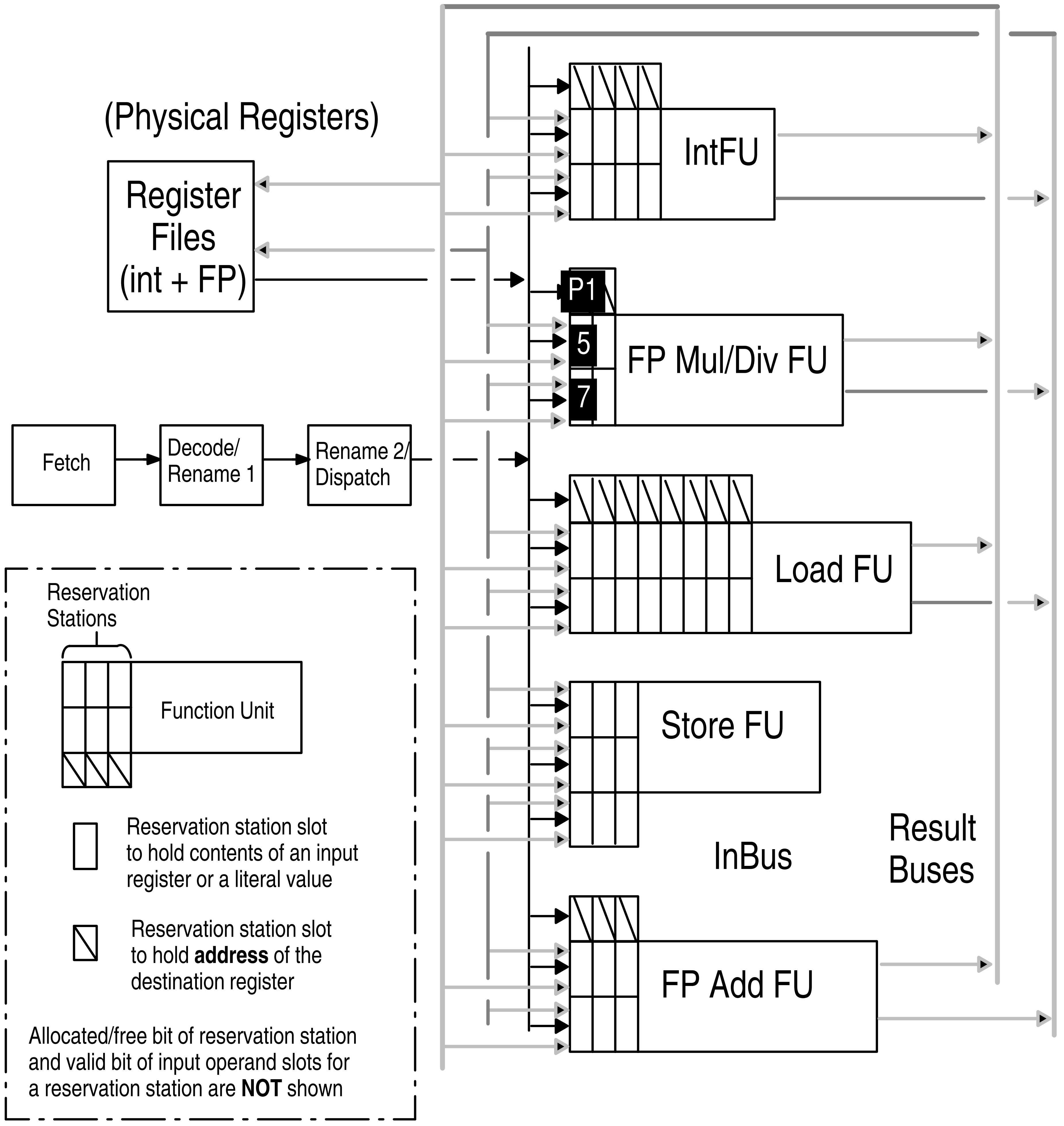

(Physical Registers)
Register Files (int + FP)
Fetch
Decode/ Rename 1
Rename 2/ Dispatch
IntFU
FP Mul/Div FU
Load FU
Store FU
FP Add FU
InBus
Result Buses
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold **address** of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are **NOT** shown

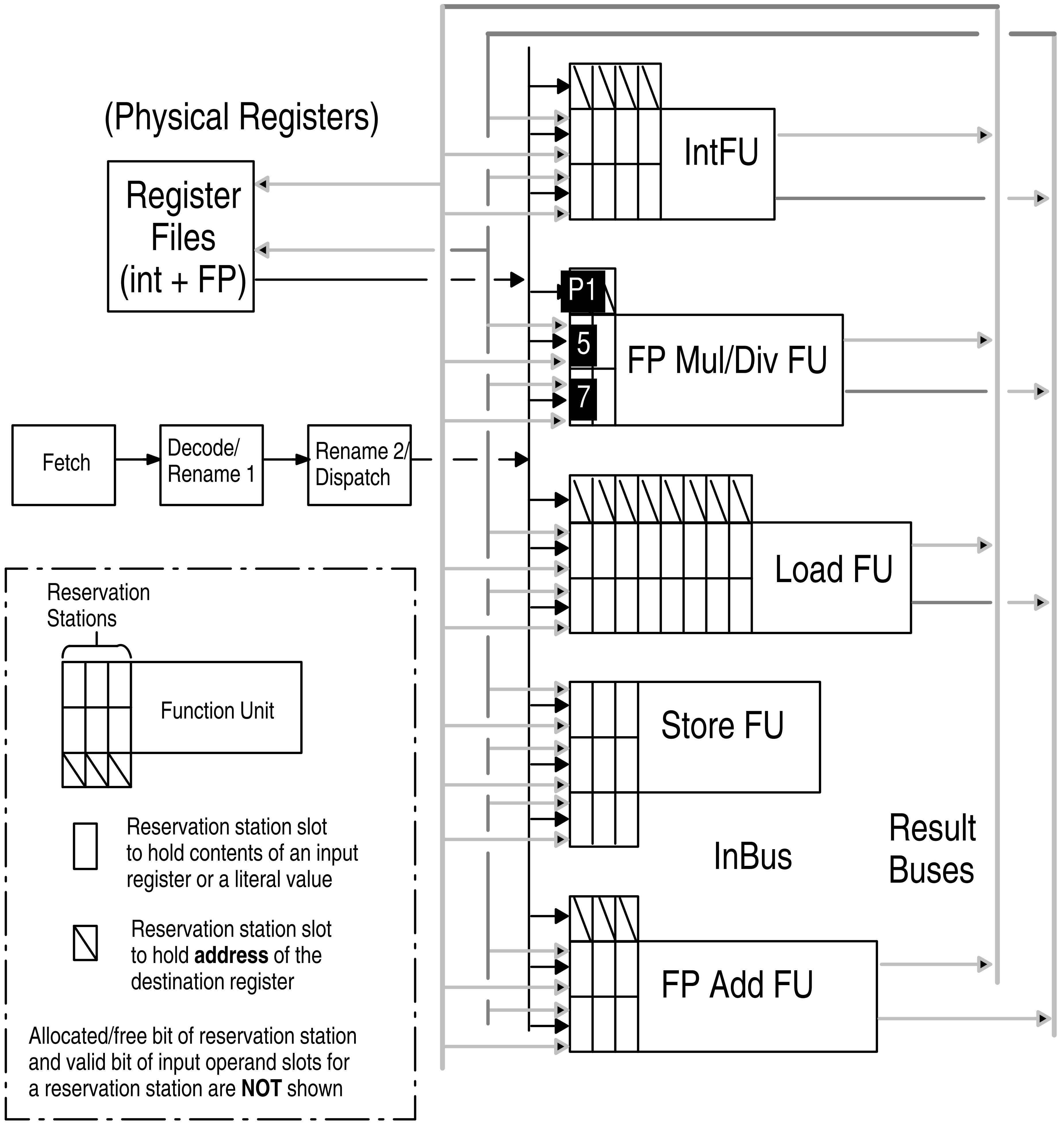

(Physical Registers)
Register Files (int + FP)
FMUL R1,R2,R3
Decode/ Rename 1
Rename 2/ Dispatch
IntFU
FP Mul/Div FU
Load FU
Store FU
FP Add FU
InBus
Result Buses
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold **address** of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are **NOT** shown

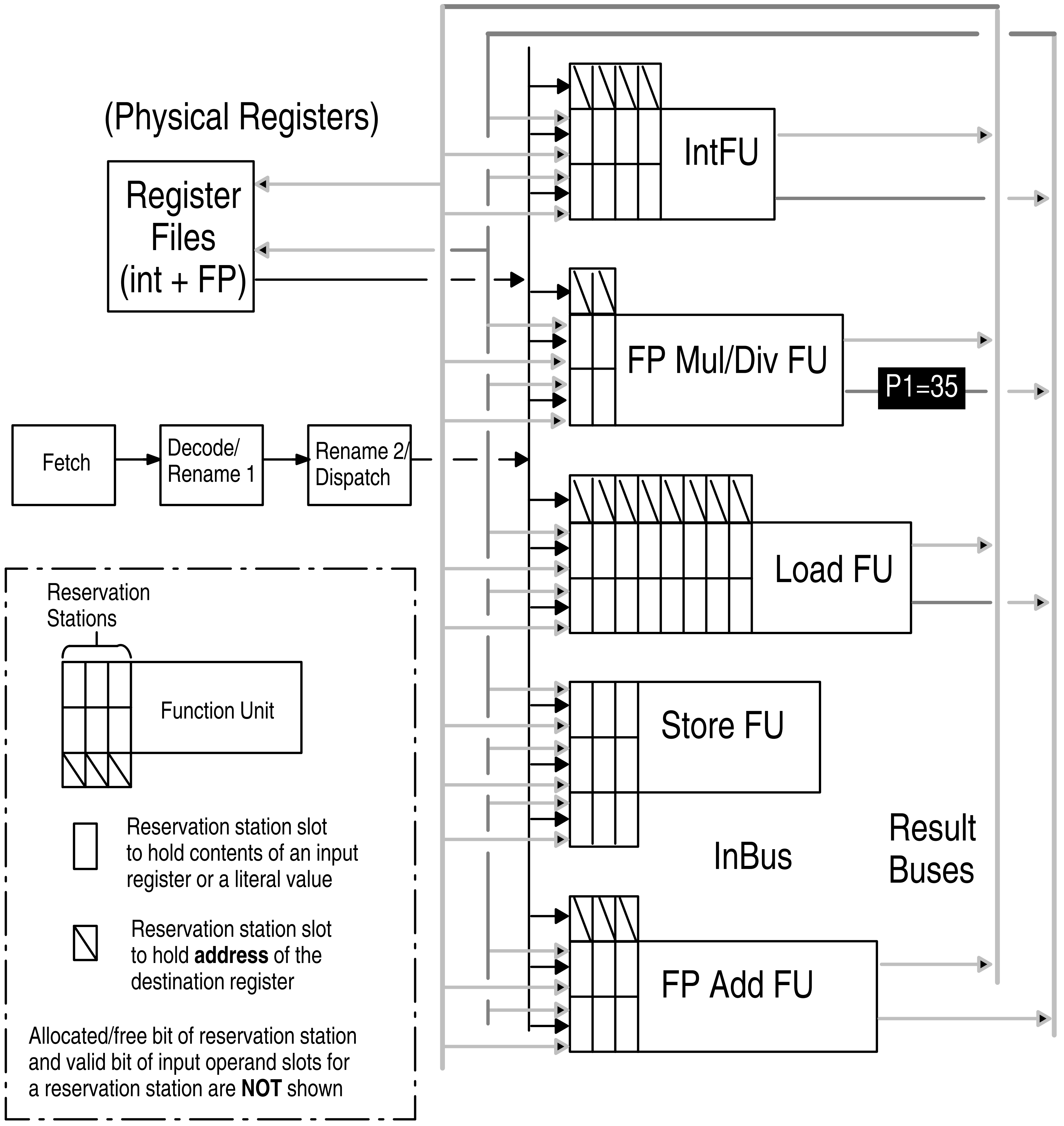

(Physical Registers)
Register Files (int + FP)
Fetch
Decode/ Rename 1
FMUL P1,P2,P3
IntFU
FP Mul/Div FU
Load FU
Store FU
FP Add FU
InBus
Result Buses
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold **address** of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are **NOT** shown

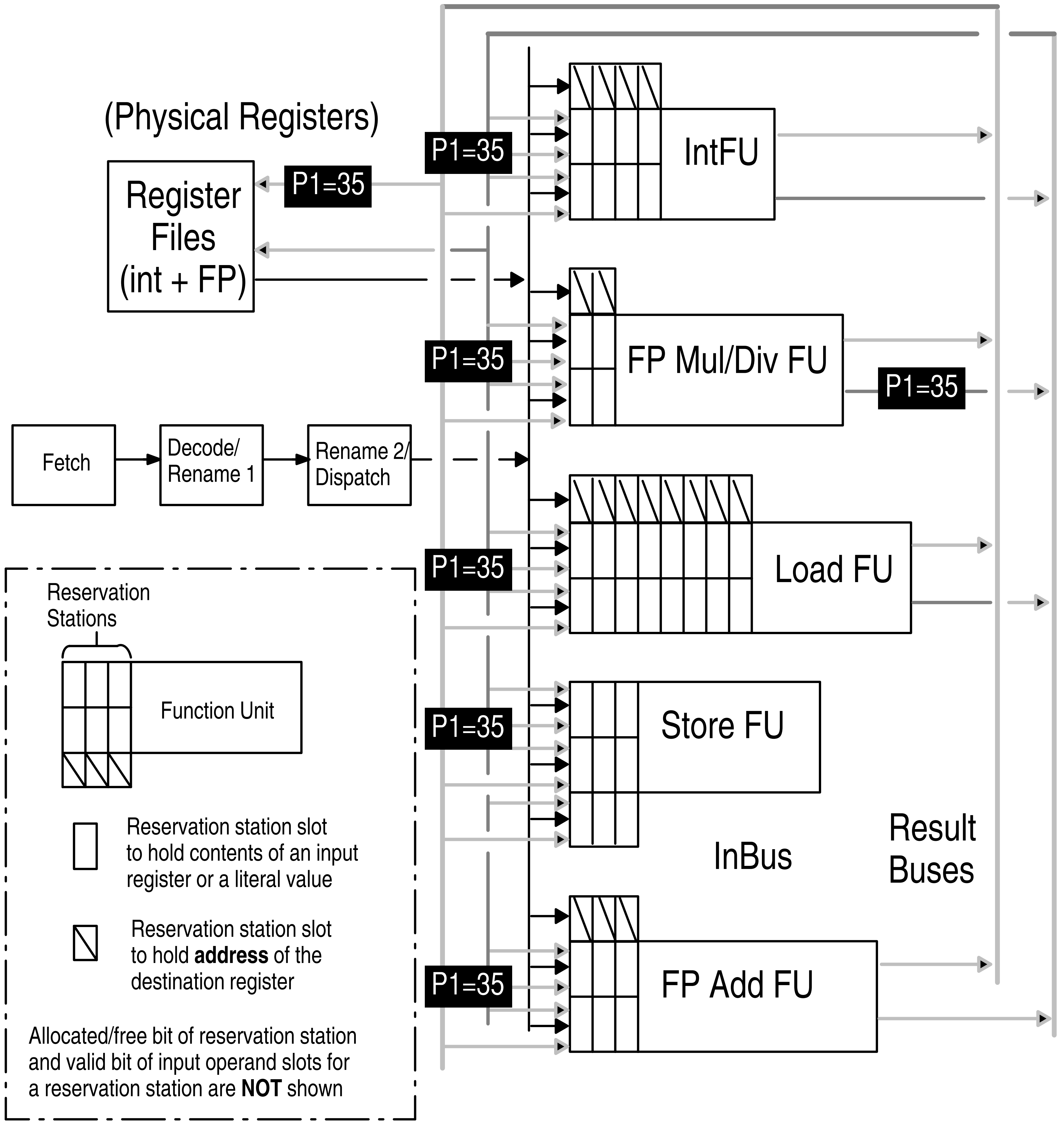


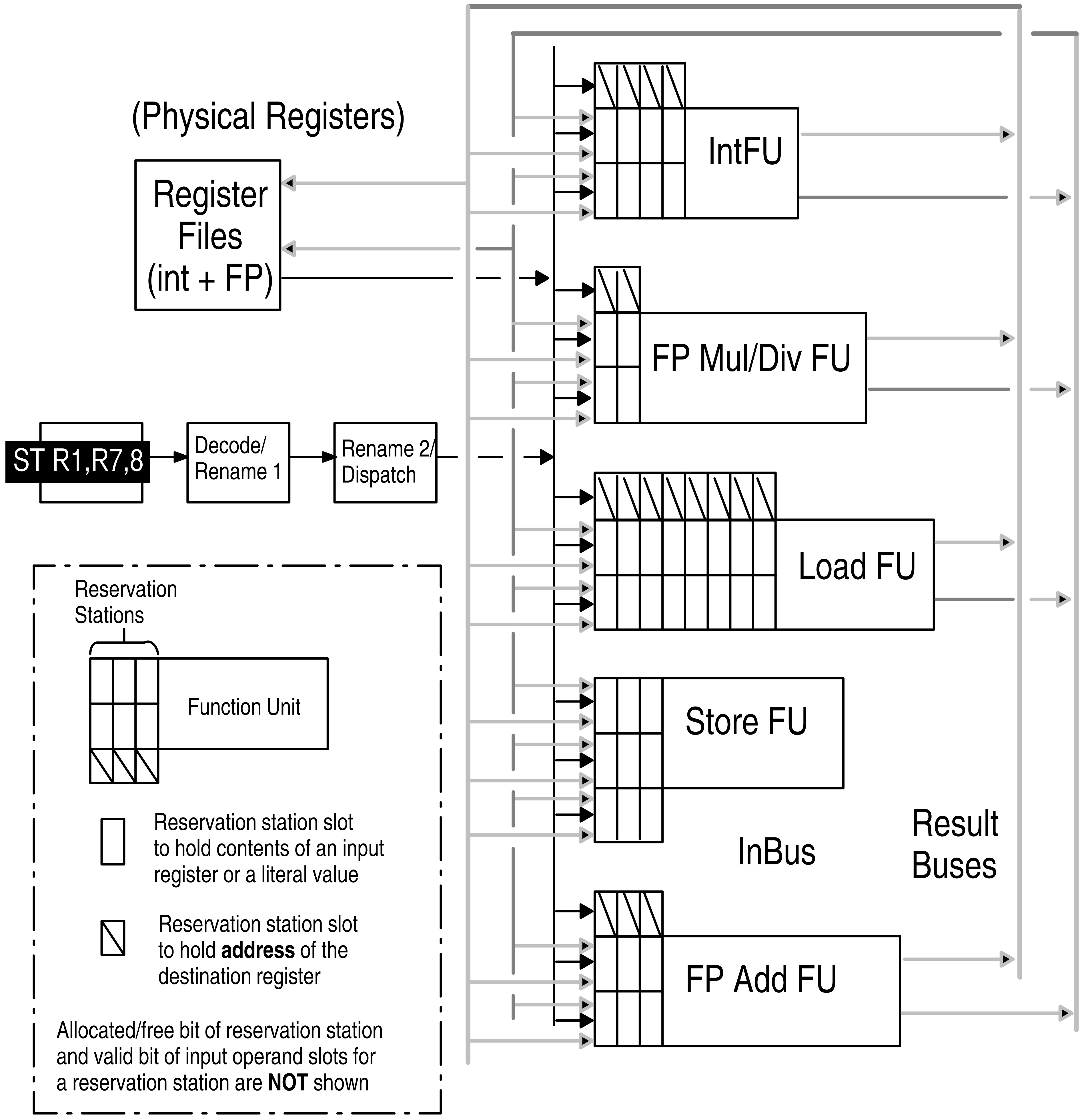
(Physical Registers)
Register Files (int + FP)
IntFU
P1
P2
P3
FP Mul/Div FU
Fetch
Decode/ Rename 1
Rename 2/ Dispatch
Load FU
Store FU
FP Add FU
InBus
Result Buses
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold **address** of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are **NOT** shown


(Physical Registers)
Register Files (int + FP)
Fetch
Decode/ Rename 1
Rename 2/ Dispatch
IntFU
P1
5
7
FP Mul/Div FU
Load FU
Store FU
FP Add FU
InBus
Result Buses
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold **address** of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are **NOT** shown

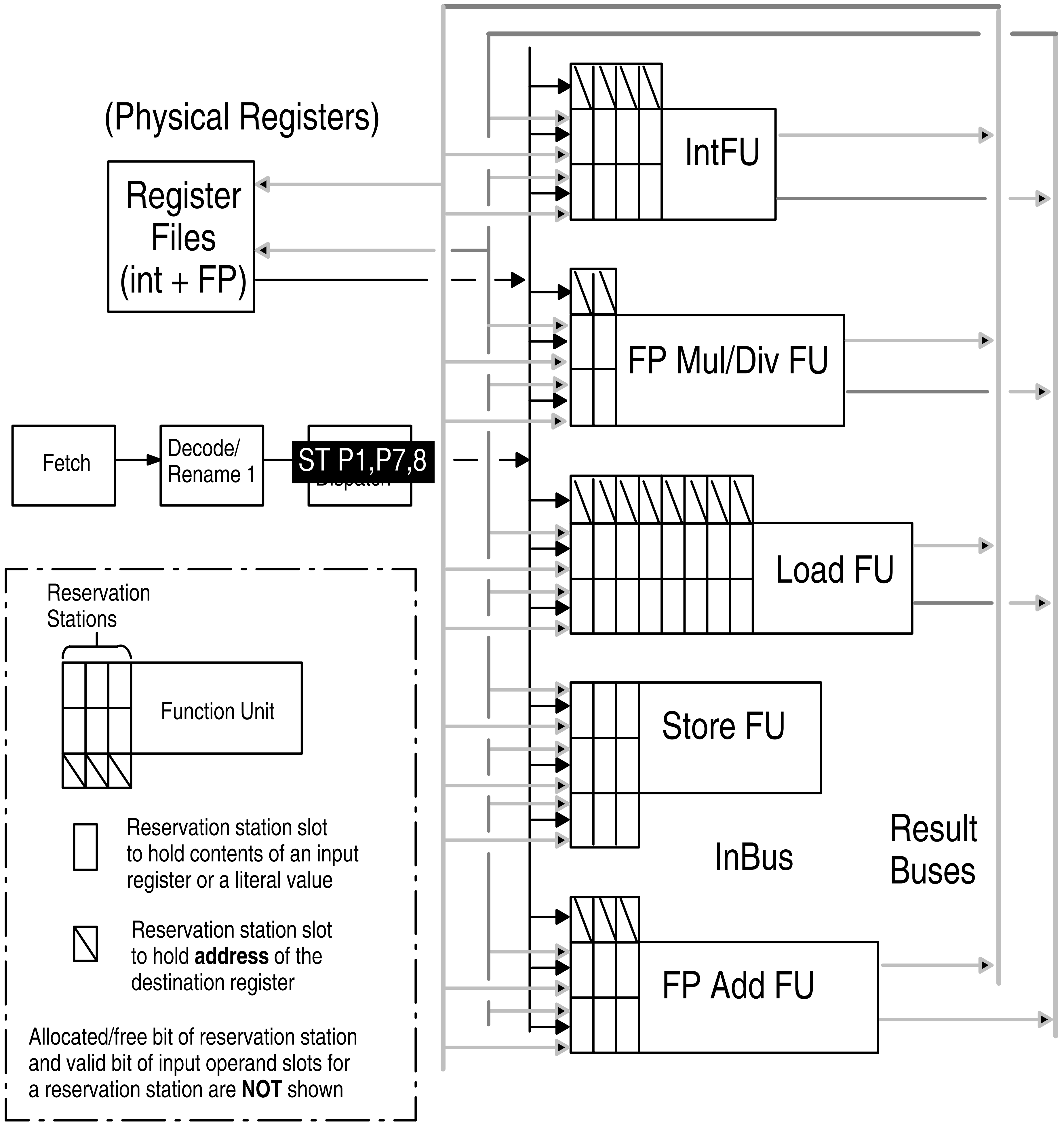

(Physical Registers)
Register Files (int + FP)
IntFU
P1
5
7
FP Mul/Div FU
Fetch
Decode/ Rename 1
Rename 2/ Dispatch
Load FU
Store FU
InBus
Result Buses
FP Add FU
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold **address** of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are **NOT** shown

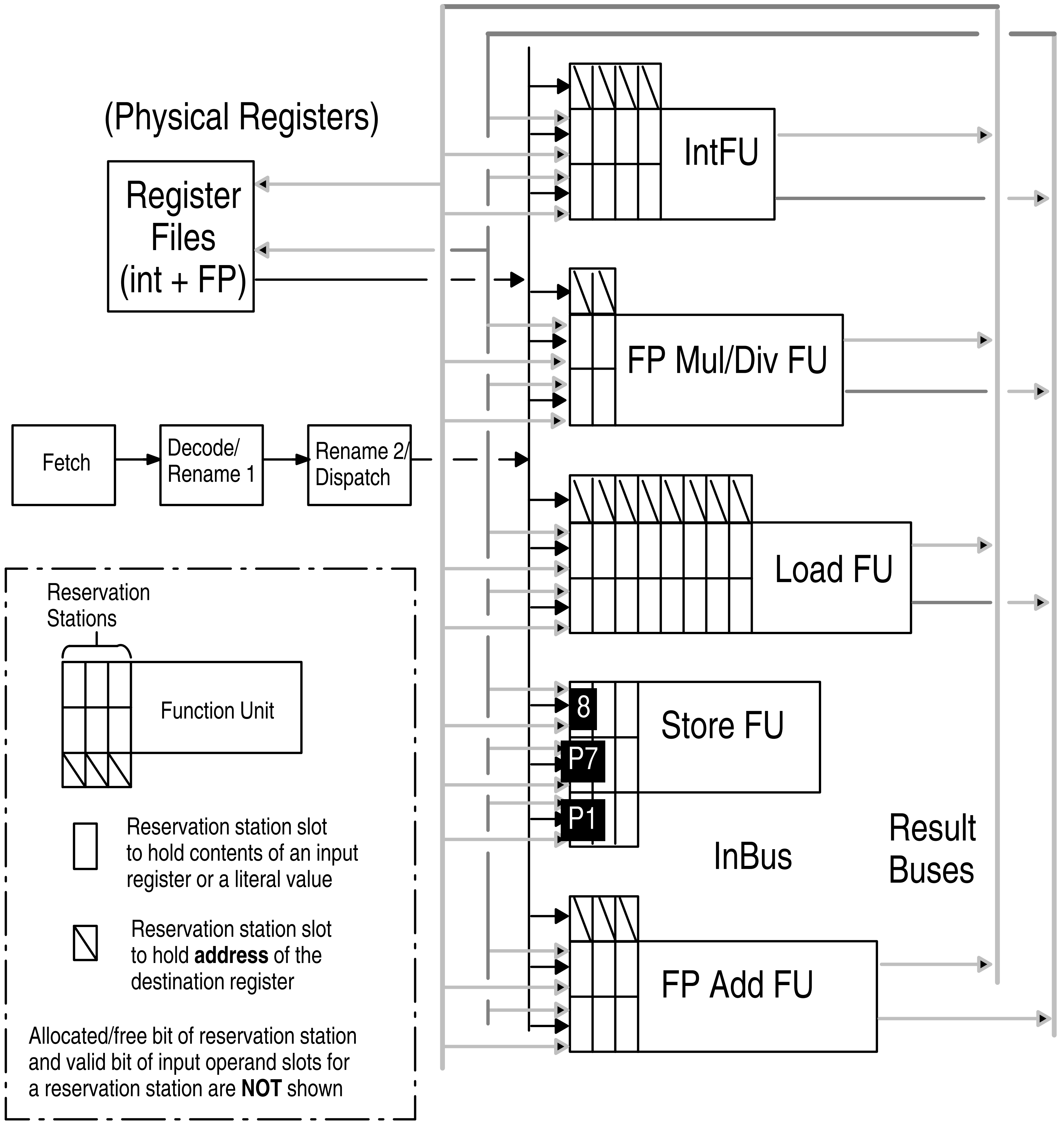

(Physical Registers)
Register Files (int + FP)
IntFU
FP Mul/Div FU
P1=35
Fetch
Decode/ Rename 1
Rename 2/ Dispatch
Load FU
Store FU
InBus
Result Buses
FP Add FU
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold **address** of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are **NOT** shown


(Physical Registers)
Register Files (int + FP)
P1=35
P1=35
IntFU
P1=35
FP Mul/Div FU
P1=35
Fetch
Decode/ Rename 1
Rename 2/ Dispatch
P1=35
Load FU
P1=35
Store FU
P1=35
FP Add FU
InBus
Result Buses
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold address of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are NOT shown

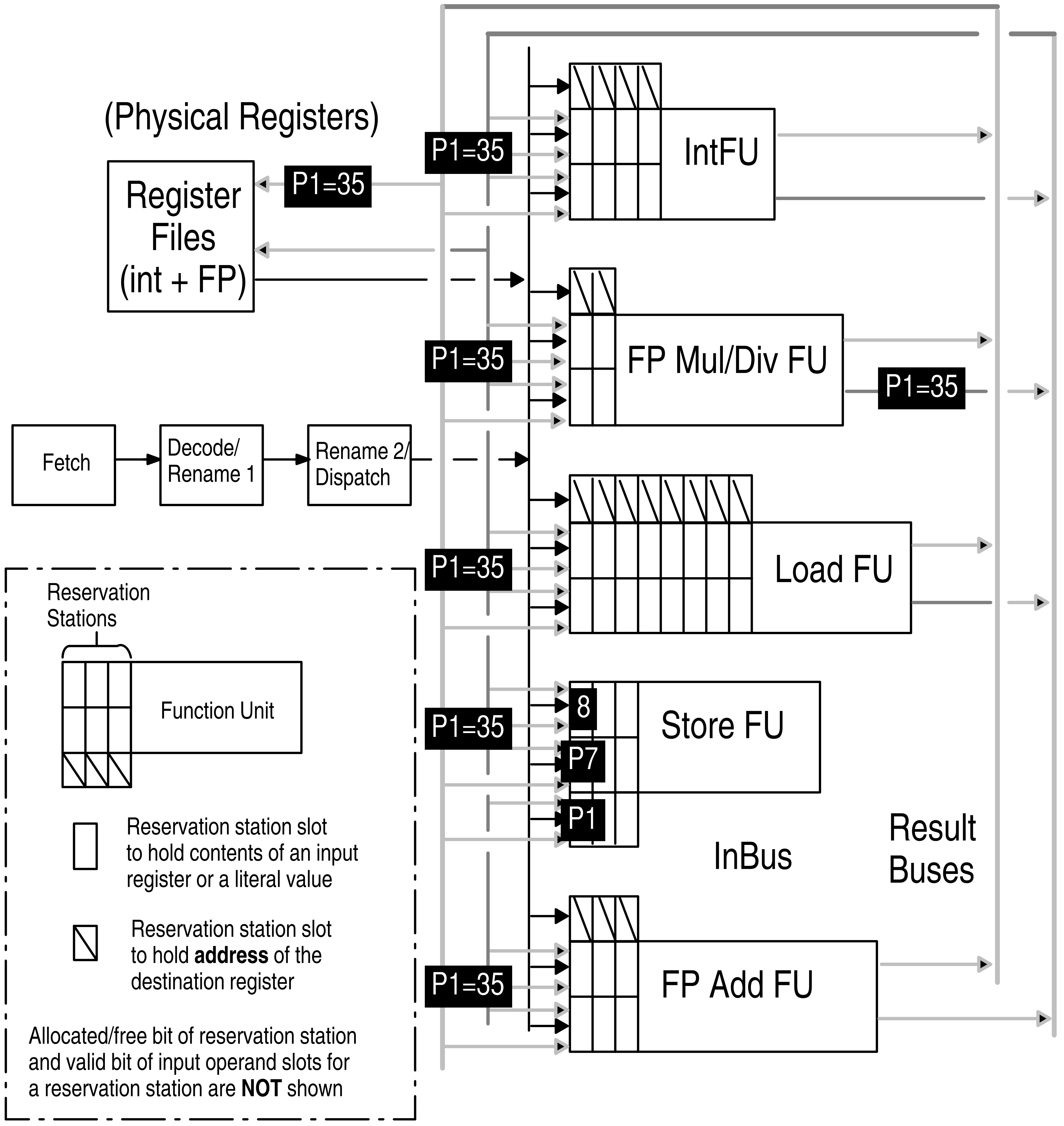


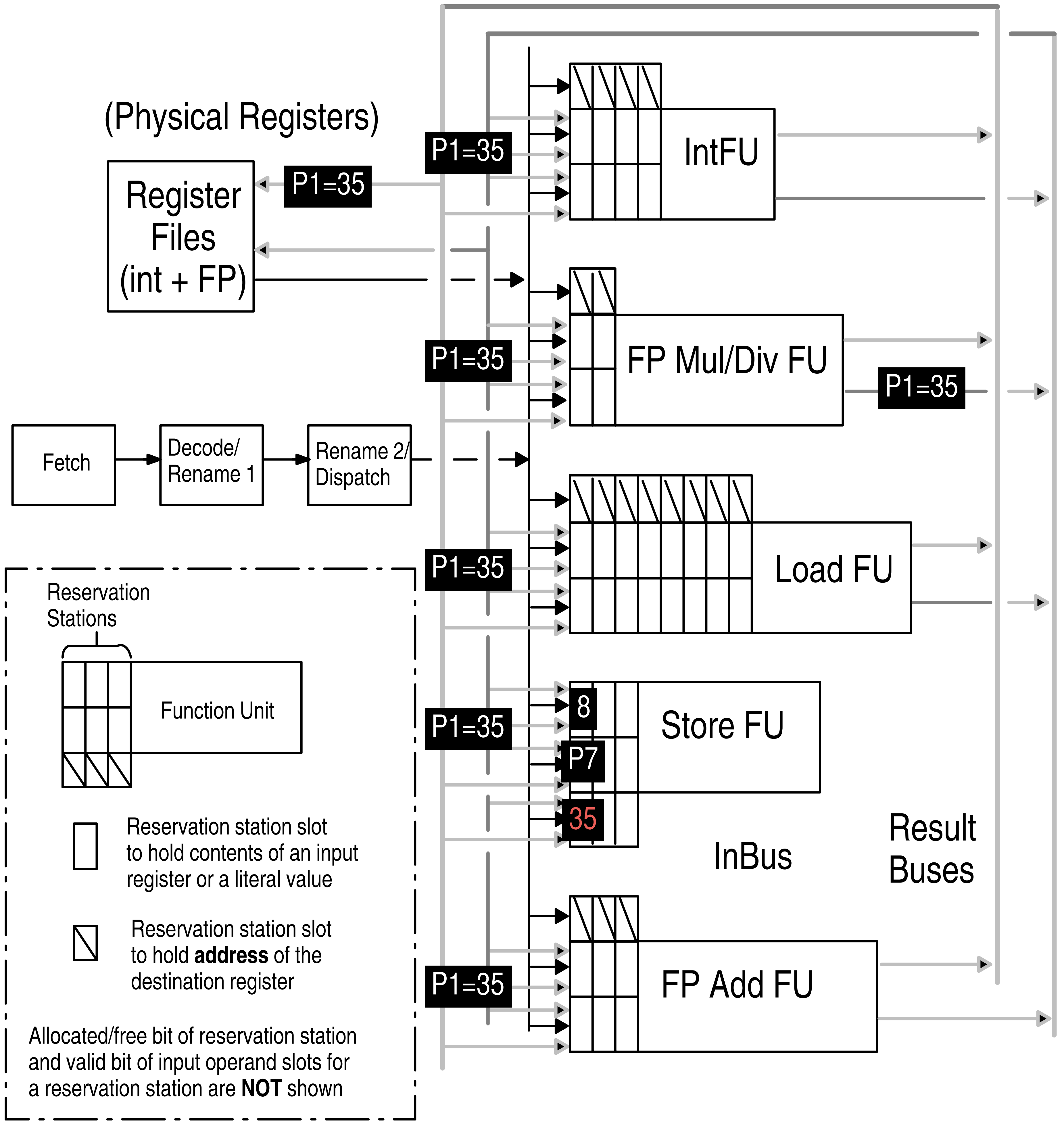
(Physical Registers)
Register Files (int + FP)
ST R1,R7,8
Decode/ Rename 1
Rename 2/ Dispatch
IntFU
FP Mul/Div FU
Load FU
Store FU
FP Add FU
InBus
Result Buses
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold **address** of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are **NOT** shown

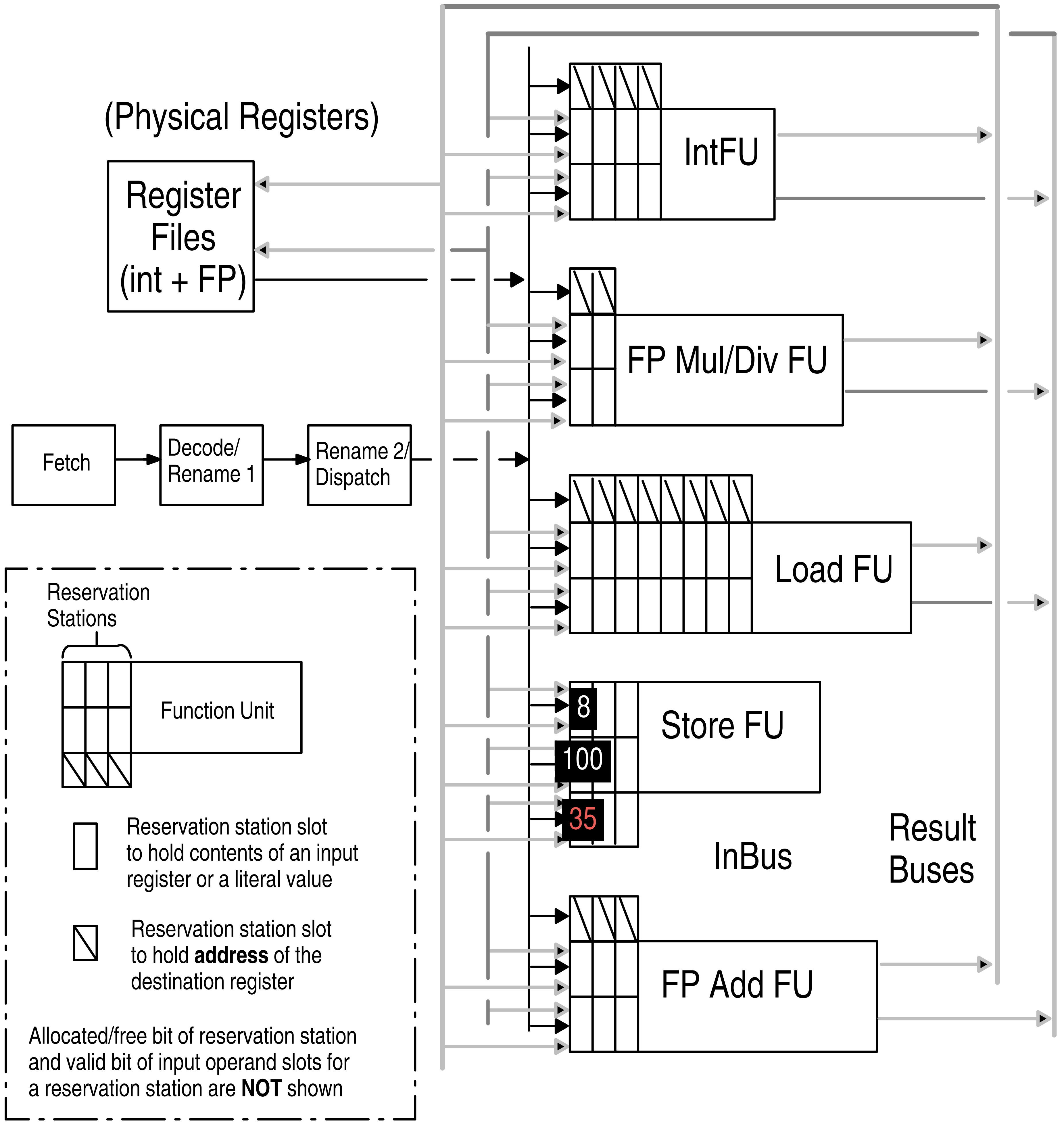

(Physical Registers)
Register Files (int + FP)
Fetch
Decode/ Rename 1
ST P1,P7,8
Dispatch
IntFU
FP Mul/Div FU
Load FU
Store FU
FP Add FU
InBus
Result Buses
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold **address** of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are **NOT** shown

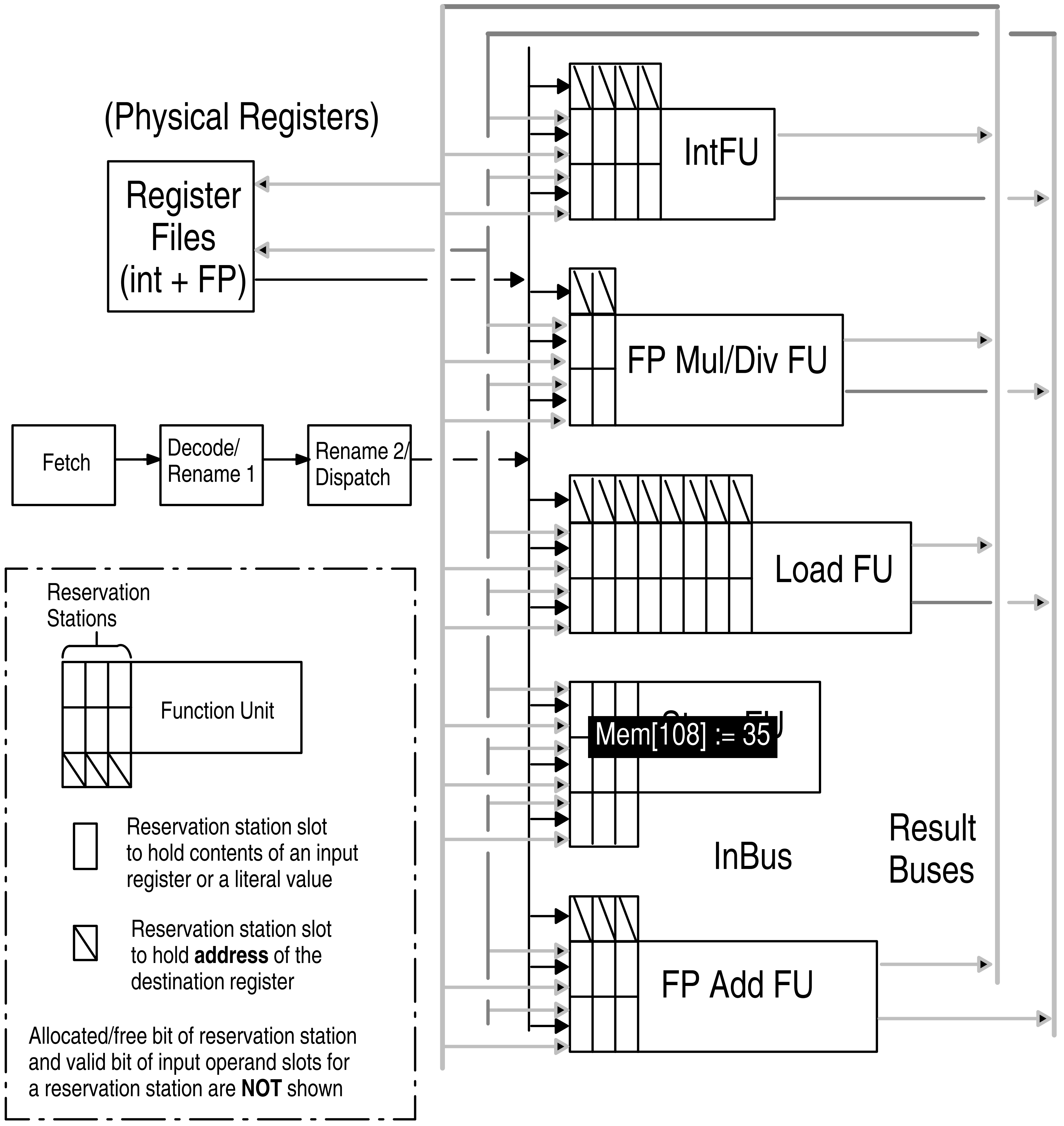


(Physical Registers)
Register Files (int + FP)
IntFU
FP Mul/Div FU
Fetch
Decode/ Rename 1
Rename 2/ Dispatch
Load FU
Store FU
8
P7
P1
InBus
Result Buses
FP Add FU
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold address of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are NOT shown


(Physical Registers)
Register Files (int + FP)
P1=35
P1=35
IntFU
P1=35
FP Mul/Div FU
P1=35
Fetch
Decode/ Rename 1
Rename 2/ Dispatch
P1=35
Load FU
Reservation Stations
Function Unit
P1=35
8
P7
P1
Store FU
InBus
Result Buses
P1=35
FP Add FU
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold address of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are NOT shown


(Physical Registers)
Register Files (int + FP)
P1=35
Fetch
Decode/ Rename 1
Rename 2/ Dispatch
IntFU
FP Mul/Div FU
Load FU
Store FU
FP Add FU
8
P7
35
InBus
Result Buses
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold **address** of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are **NOT** shown


(Physical Registers)
Register Files (int + FP)
Fetch
Decode/ Rename 1
Rename 2/ Dispatch
IntFU
FP Mul/Div FU
Load FU
Store FU
8
100
35
FP Add FU
InBus
Result Buses
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold **address** of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are **NOT** shown


(Physical Registers)
Register Files (int + FP)
Fetch
Decode/ Rename 1
Rename 2/ Dispatch
IntFU
FP Mul/Div FU
Load FU
Store FU
Mem[108] := 35
FP Add FU
InBus
Result Buses
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold **address** of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are **NOT** shown

How do we implement all of this?

- The FUs have the same characteristics that we assumed for APEX incorporating Tomasulo's technique: the physical FUs are **pipelined** and have the following latencies:

Physical FU	Latency (in cycles)
Integer FU	1
Load FU	3*
Store FU	3*
FP Add FU	4
FP Mul./Div. FU	7

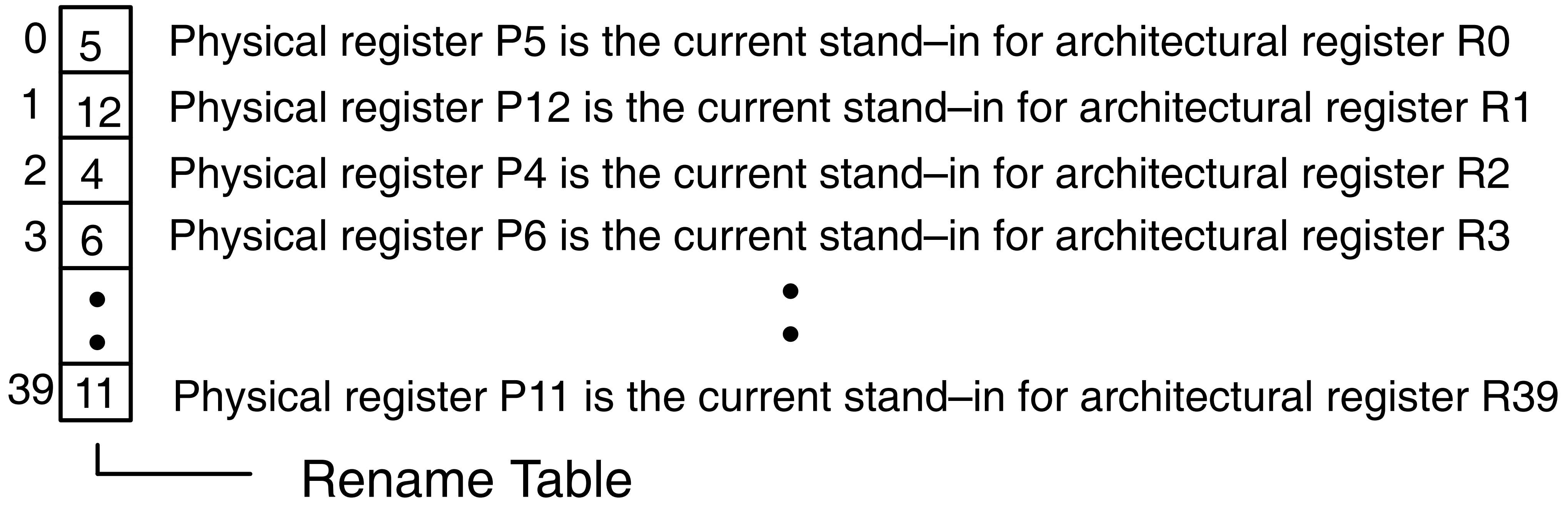
* If memory target is in cache; longer otherwise

- Assume that a single physical register file implements both integer and FP registers. Architectural registers have addresses in contiguous, neighboring ranges for integer and FP registers.

Example:

40 architectural registers: first 32 are for integers (R0 through R31), next 8 are for FP operands (F0 through F7, which are aliased to R32 to R39). This allows an unique index in the range 0 to 39 to be used to look up the rename table, as explained later.

- A rename table, indexed by the address of an architectural register. The corresponding entry in the rename table gives the physical register acting as the *stand–in* for the architectural register:


0
5
Physical register P5 is the current stand–in for architectural register R0
1
12
Physical register P12 is the current stand–in for architectural register R1
2
4
Physical register P4 is the current stand–in for architectural register R2
3
6
Physical register P6 is the current stand–in for architectural register R3
39
11
Physical register P11 is the current stand–in for architectural register R39
Rename Table

The number of entries in the rename table = the # of architectural registers

- An allocation list, *AL* [ ], of physical registers, implemented as a bit vector: this list is used for allocating free physical registers as the stand–ins for the destination architectural registers at the time of instruction dispatch:

0	1	2	3	4	5	6	7	8	9	10	11		49	50
1	1	1	0	1	0	0	1	1	1	1	1		0	0

Physical registers P0, P1, P2, P4, P7 through P11 are allocated
Physical registers P3, P5, P6, P49 and P50 are free

- An array *Renamed* [ ]: *Renamed* [k] indicates if the architectural register that physical register Pk was a stand–in for has been renamed in the rename table. This information is used for deallocating a physical register when it is updated. (*Renamed* [k] = 1 means Pk was renamed etc.)

- Status information for each physical register:

  – A bit vector, *Status* [ ] indicating if a physical register contains valid data or not: the validity information is obtained by looking at *AL* [ ] and *Status* [ ] in conjunction: A physical register Pj contains valid data if Pj is allocated (i.e., *AL*[j] = 1) and if *Status* [j] is a one.

- Forwarding information that allows the simultaneous update of a physical register by a VFU and the forwarding of the result being written to the physical register to waiting VFUs:

  – A bit vector *Wk* for *each physical register* Pk that indicates the VFU slots that are awaiting the data to be written into this physical register: *Wk*[j] is set to 1 if the result to be written to Pk is to be forwarded to slot #j

  – Each VFU slot that can have a data forwarded to it is given a unique id

- A bit vector indicating the allocation status of each VFU, VFU_Status [ ].

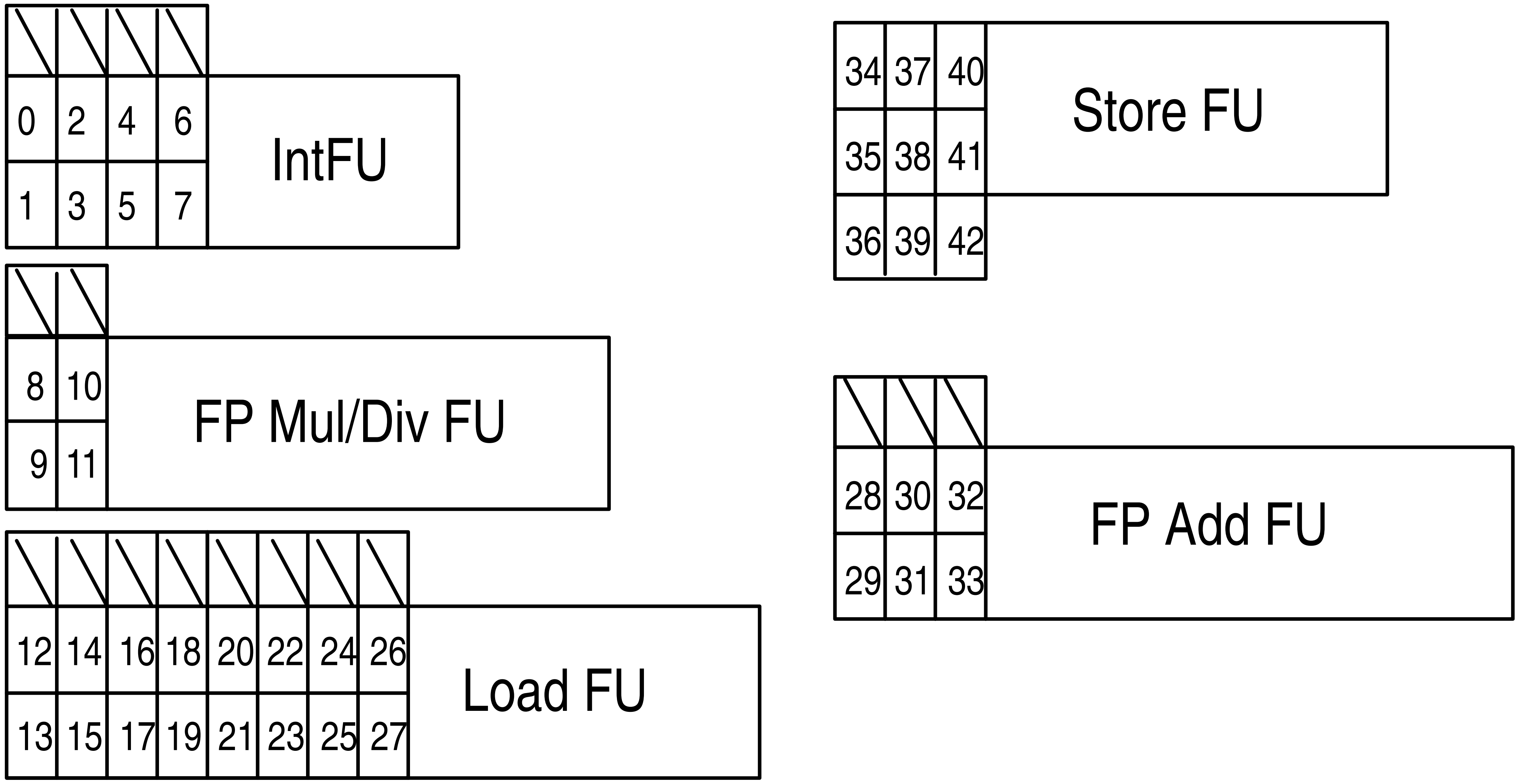
– All of these status information are maintained in a scoreboard–like structure.

W Matrix

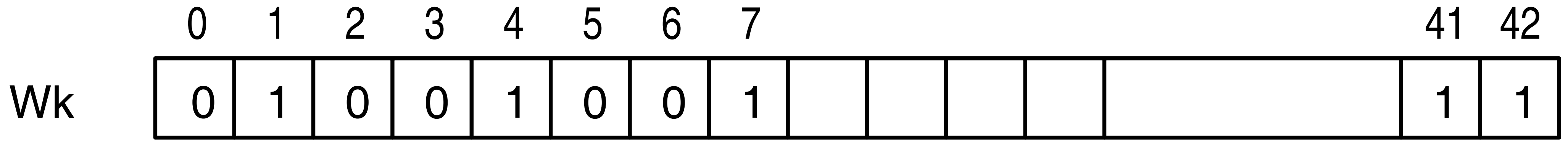
Dependent Resource

Physical Register

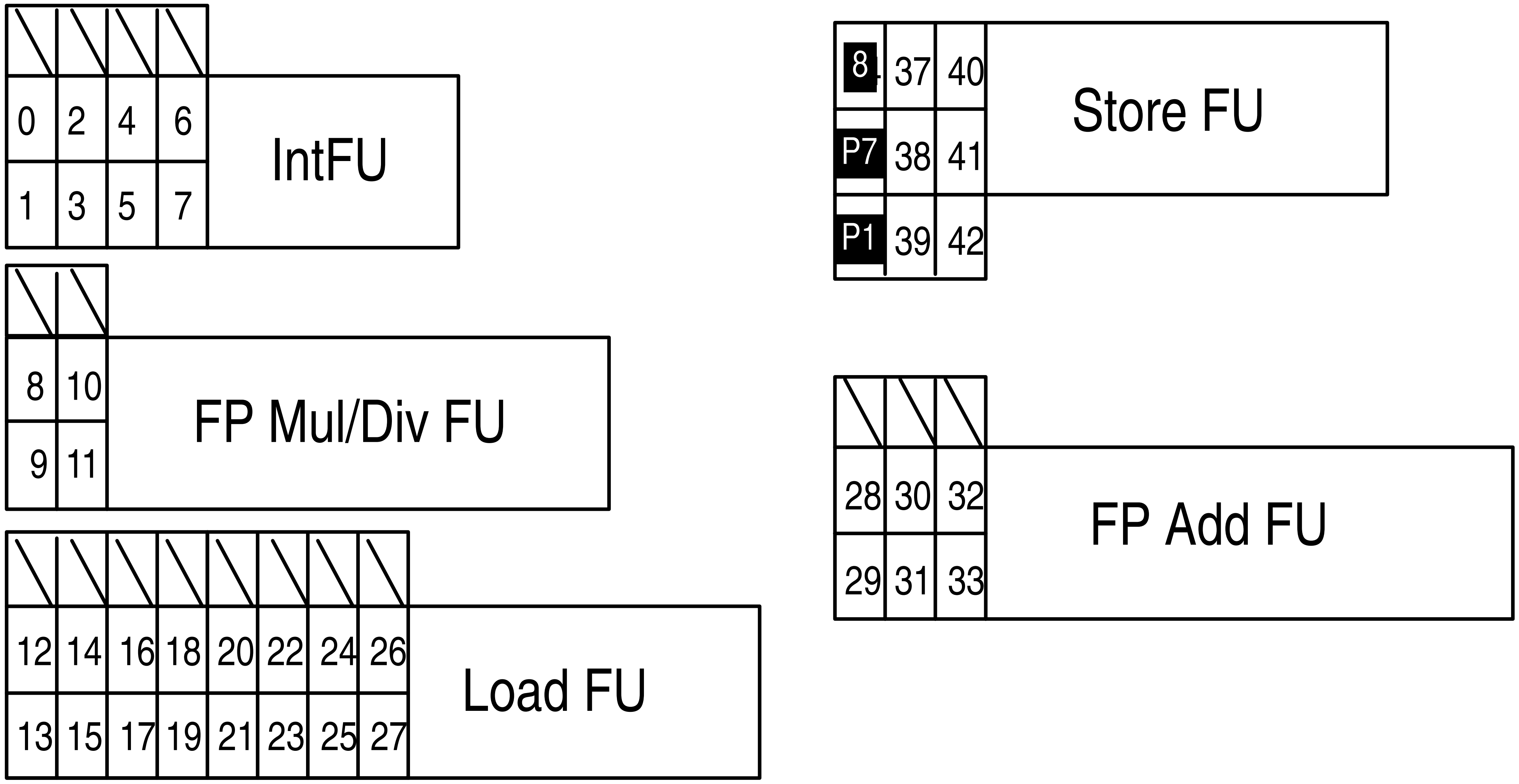
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0															
1															
2															
3															
4															
5															
6															
7															
8															
9															
10															
11															
12															
13															
14															
15															



VFU Slot Numbering



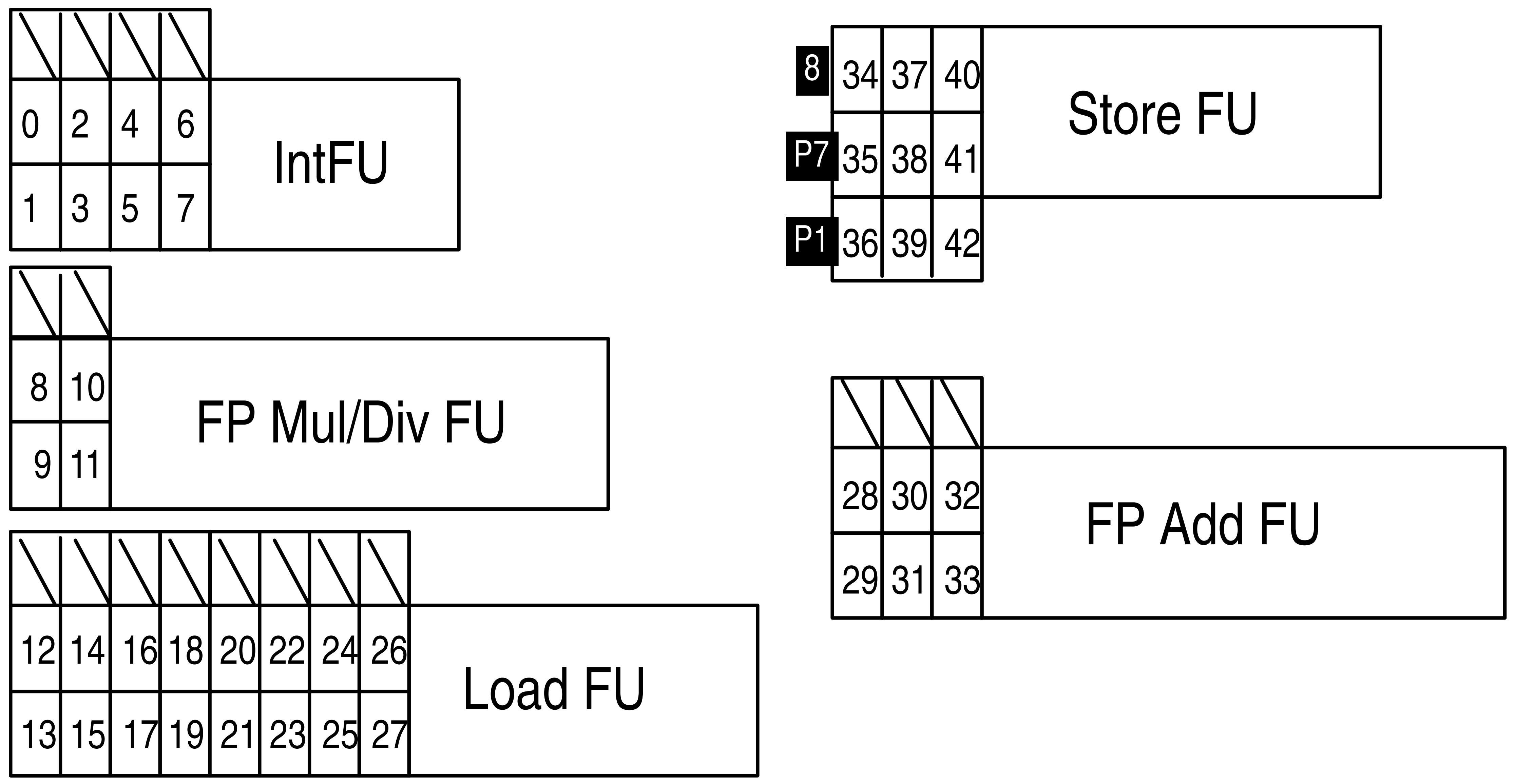
Data to be written to Pk has to be forwarded to VFU slots numbered 1, 4, 7, 41, 42



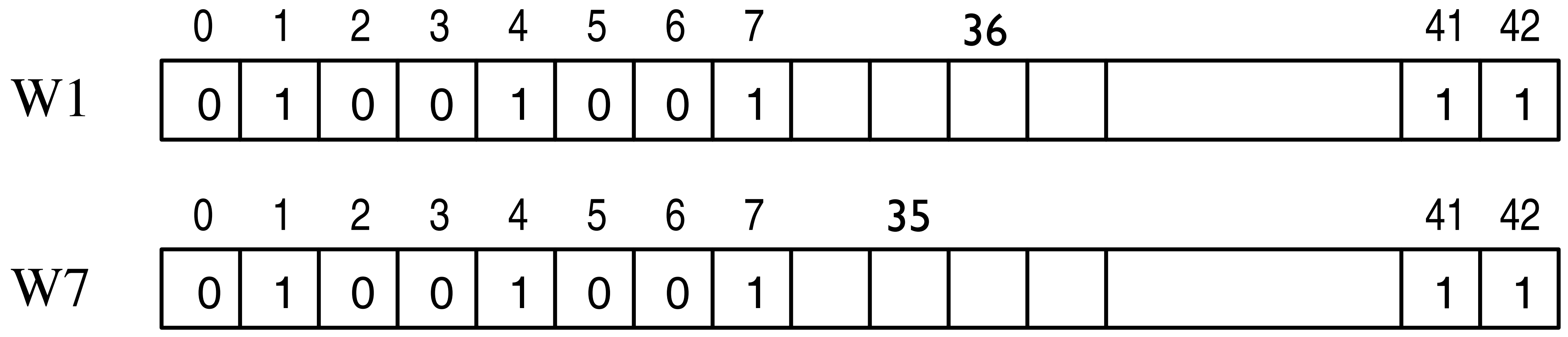
VFU Slot Numbering

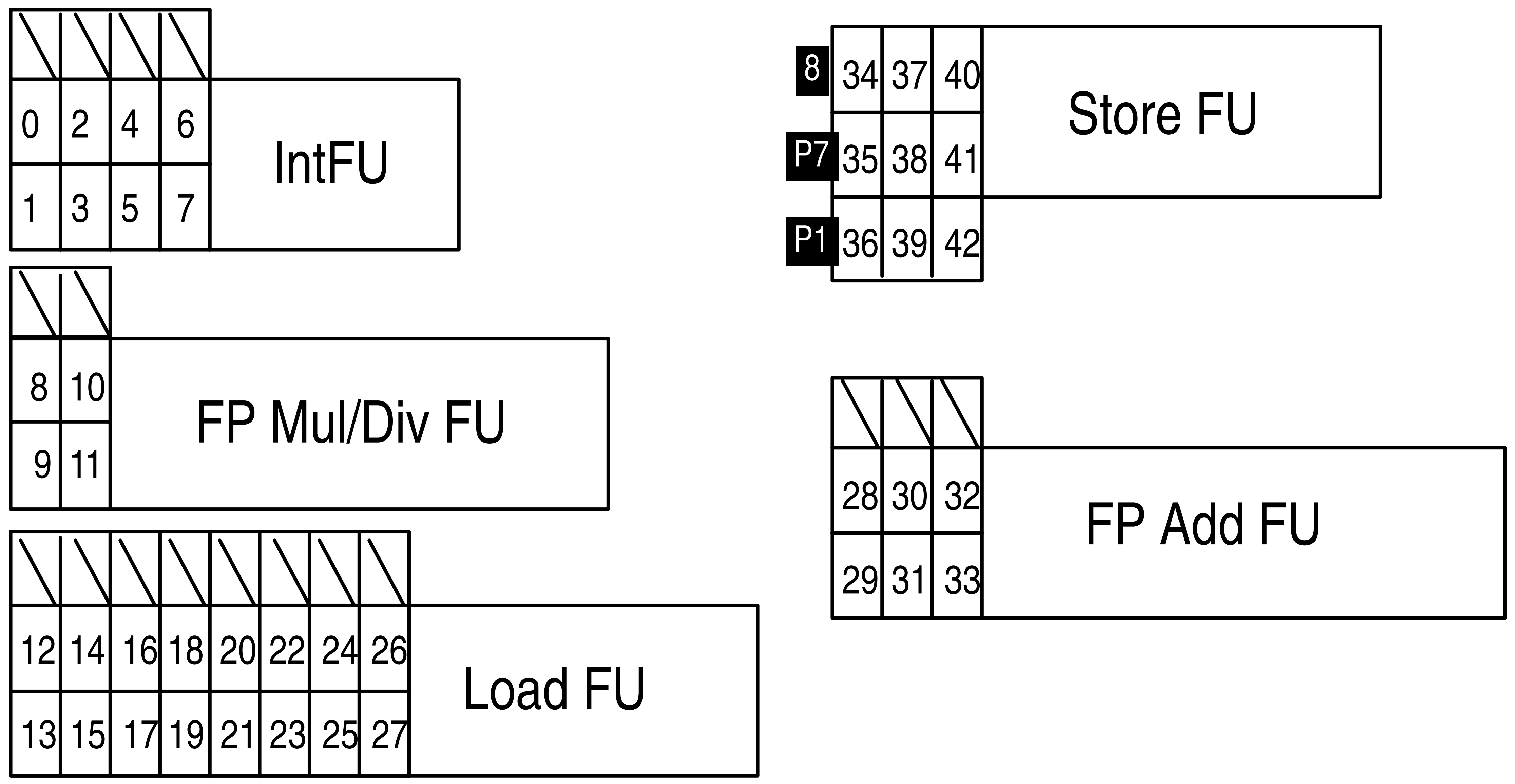
	0	1	2	3	4	5	6	7						41	42
Wk	0	1	0	0	1	0	0	1						1	1

Data to be written to Pk has to be forwarded to VFU slots numbered 1, 4, 7, 41, 42

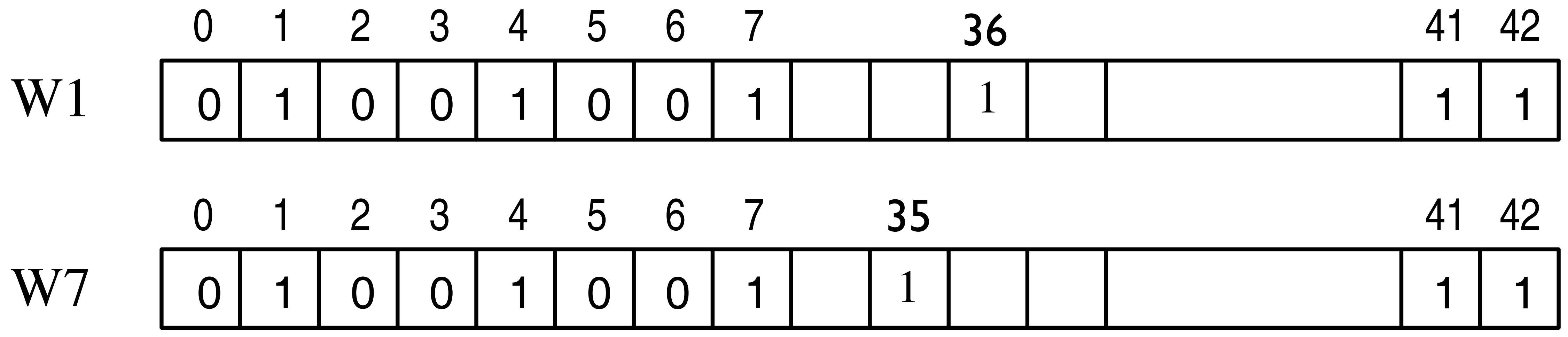


VFU Slot Numbering

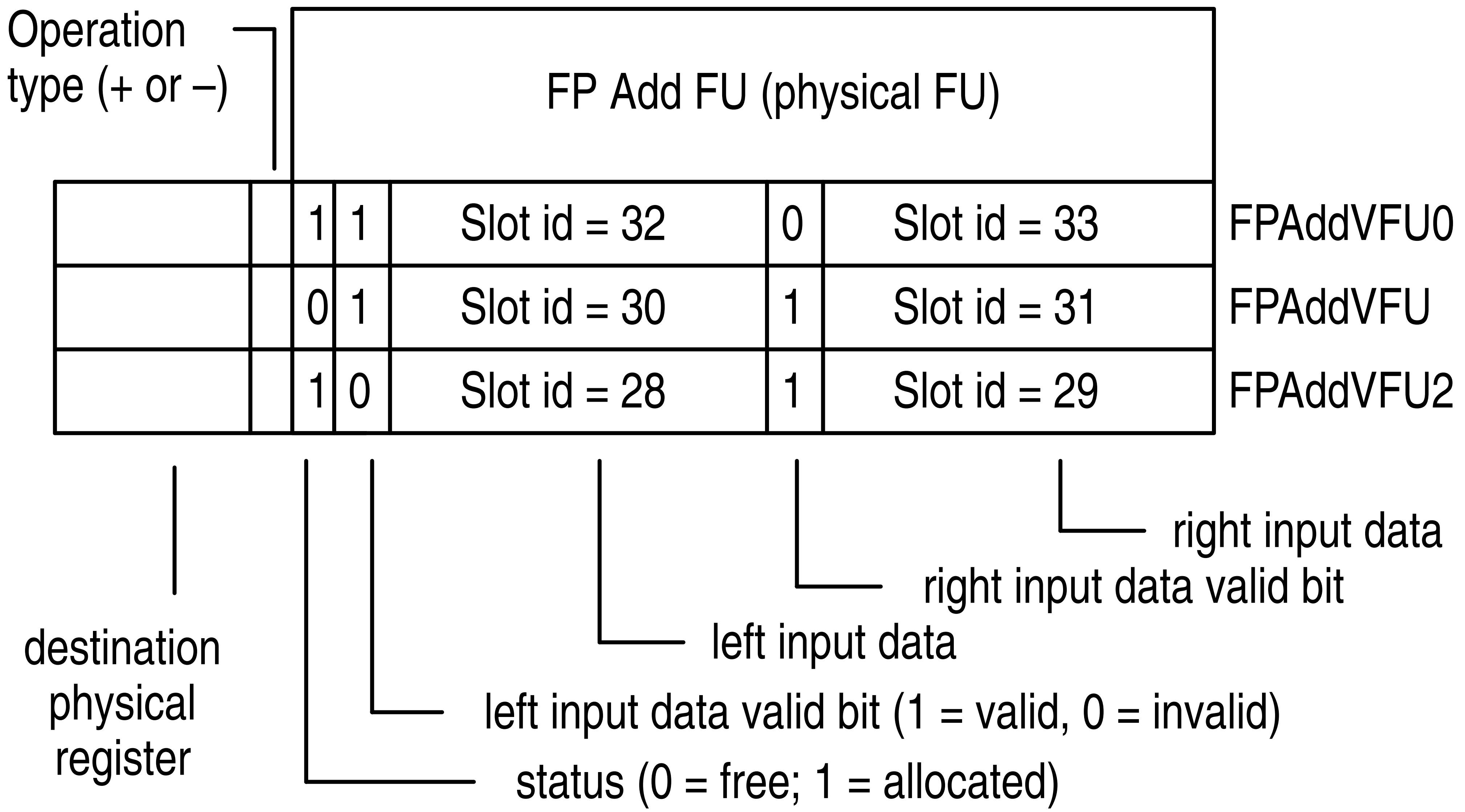




VFU Slot Numbering

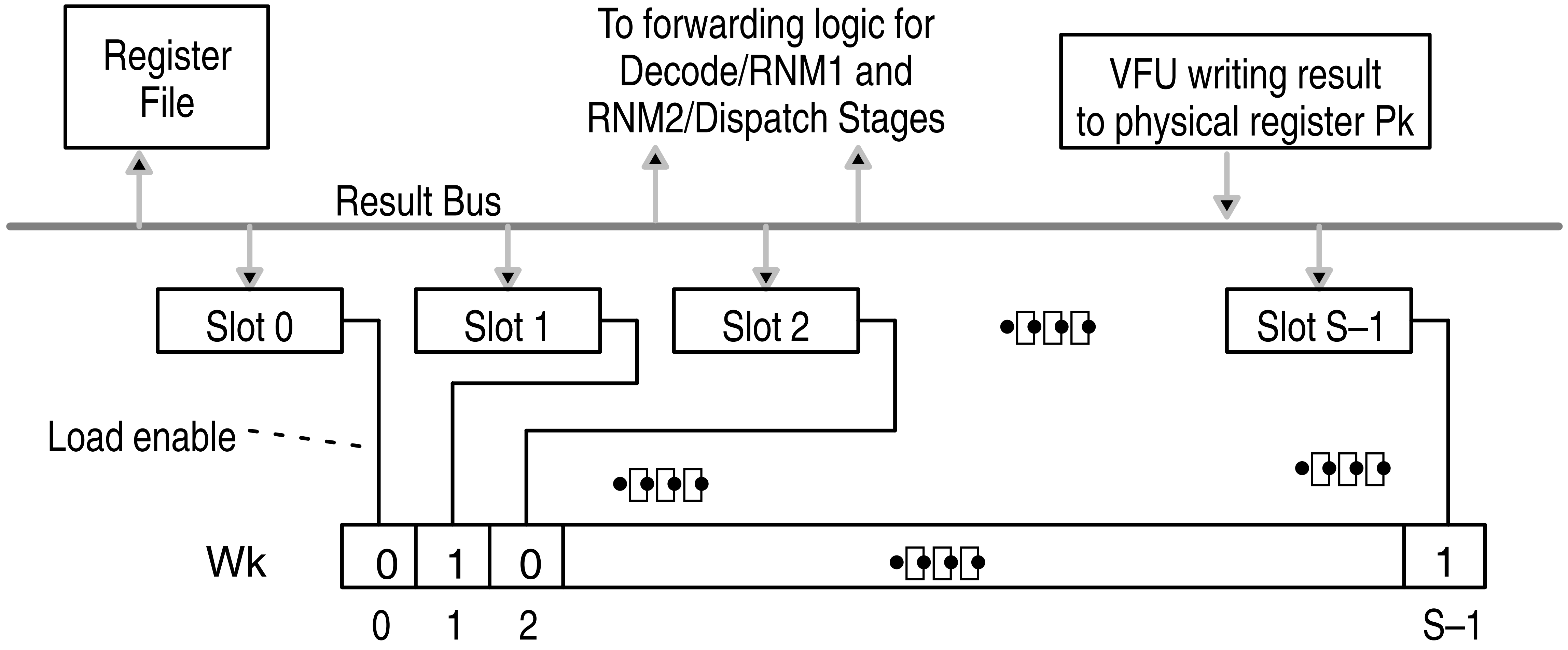


- Format of a RSEs that requires two input operands:


Operation type (+ or –)
FP Add FU (physical FU)
1 1 Slot id = 32 0 Slot id = 33 FPAddVFU0
0 1 Slot id = 30 1 Slot id = 31 FPAddVFU
1 0 Slot id = 28 1 Slot id = 29 FPAddVFU2
right input data
right input data valid bit
left input data
left input data valid bit (1 = valid, 0 = invalid)
status (0 = free; 1 = allocated)
destination physical register

  – Note need to have operation type field with some RSEs (e.g., integer VFUs) since they implement multiple functions.

- Data forwarding is accomplished by using the information in the W array for the physical register being written. A slot after receiving its data from the forwarding bus sets its valid bit to 1:


Register File
To forwarding logic for Decode/RNM1 and RNM2/Dispatch Stages
VFU writing result to physical register Pk
Result Bus
Slot 0
Slot 1
Slot 2
Slot S–1
Load enable
Wk
0
1
0
1
0
1
2
S–1

Using Wk to forward data being written to physical register Pk

► **After a physical register is updated, its *W* vector is cleared**

Renaming and Dispatching

- How we will label registers:
  - Rj ← Rk *op* Rl		Architectural
  - Pq ← Pr *op* Ps		Physical

Allocate or Stall

- Need:
  - Free physical register
  - Free VFU
- Stall until both are free

Aliasing and Renaming

- Read r←Alias[k] and s←Alias[l]
- Maps Rk to Pr and Rl to Ps

- Read p←Alias[j]
- Set Renamed[p]

- Assign Alias[j]←q
- Maps newly allocated Pq to Rj
- Mark Pq as allocated: AL[q]←1

$Rj \leftarrow Rk \; \mathit{op} \; Rl$

$Pq \leftarrow Pr \; \mathit{op} \; Ps$

Resolving Dependencies

- Allocated VFU #n, which has slots
  - a and b
- Dependent on Pr and Ps
- Wr[a] $\leftarrow 1$
- Ws[b] $\leftarrow 1$

Rj $\leftarrow$ Rk *op* Rl

Pq $\leftarrow$ Pr *op* Ps

Rj ← Rk *op* Rl

Pq ← Pr *op* Ps

Substitution

- Decoded instruction has register numbers replaced with physical ones.
- Dispatch this:
  - Pq ← Pr *op* Ps

Dispatching

- If physical registers are valid, they can be read out during dispatch. Data stored in VFU.
- For invalid ones, VFU slots marked invalid; tagged with physical register number.
- Address of destination phys. register also stored in VFU.

- **VFU completion and data forwarding**:
  - When a FU completes, it arbitrates for access to any one of the two result buses. On being granted access to a result bus, it writes the result into the destination register.
  - At the same time, the Wq list associated with the destination register Pq is used to forward the data to the waiting function unit slots, as described earlier.
  - This result is also forwarded to the instructions currently within the Decode/Rename1 and Rename2/Dispatch stages.
  - If *Renamed* [q] is set, Pq is added back to the free list (i.e., *AL* [q] is reset to a 0). Otherwise, *Status* [q] is set to indicate that Pq contains valid data.
- **Deallocating a VFU:** Once the operation for a VFU has started up (and if the physical FUs are pipelined), the VFU can be de–allocated. Note contrast with Tomasulo's.

- **Deallocating physical registers**:

► A physical register can be deallocated at the time a physical register is updated, as described earlier.

► Another opportunity for deallocating a physical register occurs when an instruction renames the architectural register for which physical register Pq was the most recent instance **and** if there is no instruction waiting to read Pq (which implies that Pq's status bit should indicate Pq's content as **valid**). If these conditions are valid, Pq is marked for deallocation in the Decode/Rename1 stage but the actual deallocation does not take place *till the instruction moves into the Rename2/Dispatch stage*, one cycle later.

  – The reason for delaying the actual deletion is to allow the instruction, say I, preceding the one that marked Pq as renamed to read Pq as a possible input operand when I is in the Rename2/decode stage.

Deallocation Summary

- Can free physical register when:
  - Renamed[q] is already true and result Pq is being forwarded, OR
  - No out-standing dependencies (Pq already valid) and it is being renamed.

► **When a physical register is deallocated, its W vector is cleared**.

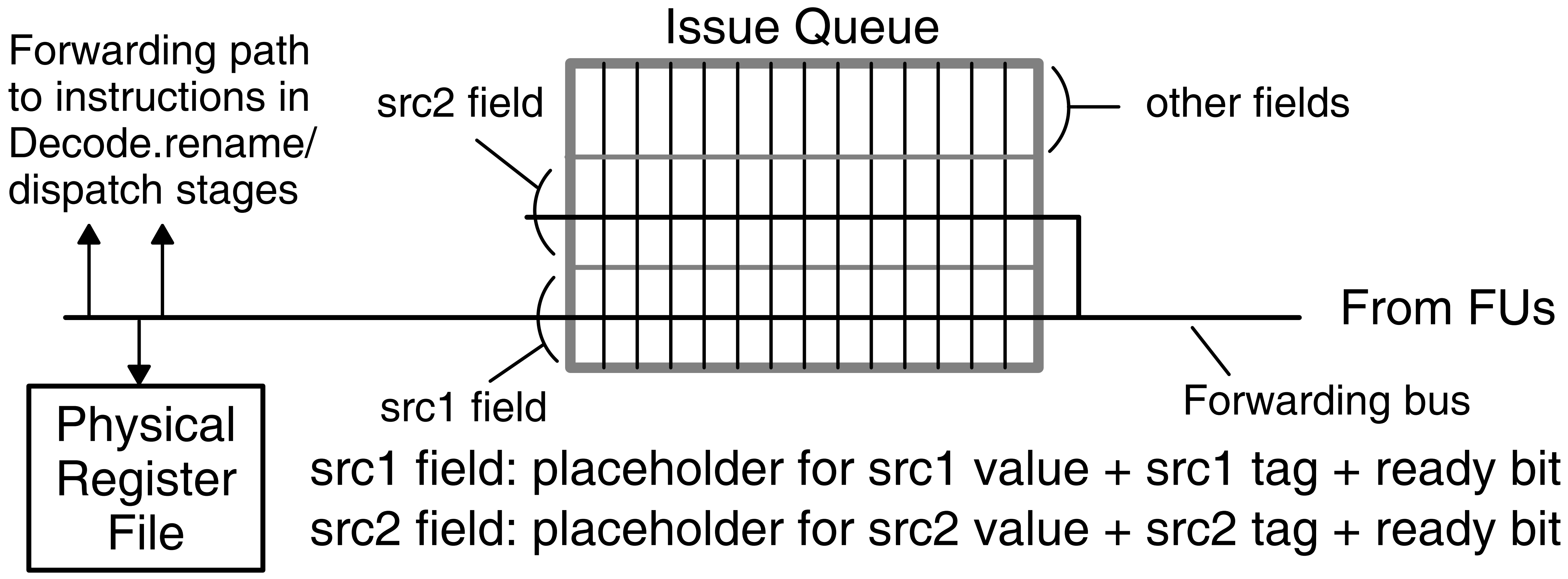
- Additional terminology related to dynamic scheduling:

  instruction wakeup: the process of enabling waiting instructions for potential execution when such instructions receive the last of any source operands they were waiting for.

  instruction selection: the process of selecting a subset of the enabled instructions for issue.
- Precise interrupts are easy to implement with register renaming (LATER). Note also the differences between register renaming and Tomasulo's algorithm (irrespective of the specific implementation of register renaming).

Alternative Implementation of Register Renaming

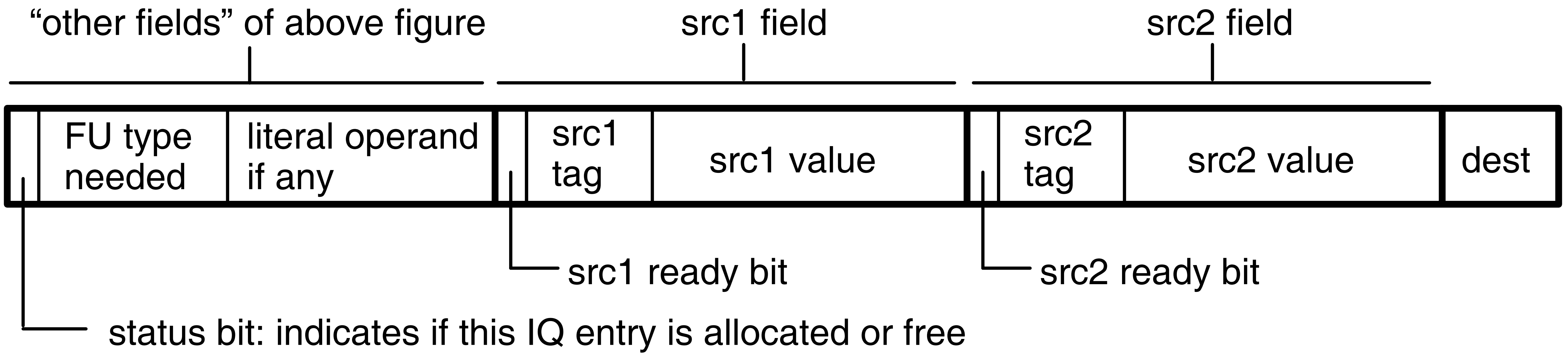
- Uses centralized Issue Queue (IQ) and associative tag matching
- Associative matching, using the address of physical register being written as key or **tag** is used to accomplish forwarding to data slots within the IQ as shown below (paths for dispatching and issuing to/from the IQ are NOT shown):



src1 field: placeholder for src1 value + src1 tag + ready bit
src2 field: placeholder for src2 value + src2 tag + ready bit

- Every physical register has a bit (**register valid** bit) that indicates if the contents of the register is valid or not.

- Format of an entry in the issue queue:



src1 tag, src2 tag = addresses of physical registers corresponding to src registers

- Instruction decoding and dispatching steps:

a. If a free IQ entry is not available OR a free physical register for the destination is unavailable, then stall. Otherwise, continue to Step b.

b. Look up rename table for physical register addresses corresponding to the source registers. Assign the physical register for the destination and update the rename table.

c. If a source register is valid (as indicated by its register valid bit), read it out into the corresponding field in the selected IQ entry and set the ready bit of this entry to mark the contents of the source field as valid.

d. If a source register is not valid, set the src register address field in the IQ entry to the physical address of the source register and clear the ready bit to indicate that the corresponding source register value is not ready.

e. If the instruction has a single source register, set the ready bit of the **unused** source register.

f. If the instruction has any literal operand, read it out into the literal field of the IQ entry. Set the "dest" field in the entry to the address of the register allocated in Step b).

h. Set the "FU type needed" field in the IQ entry to indicate the type of FU needed to execute this instruction.

i. Mark the IQ entry as allocated.

- Instruction writeback:

a. When an instruction completes, wait till a forwarding bus is available. Continue with the following steps when a forwarding bus is available. (Several FUs completing at the same time have to arbitrate for access to the forwarding buses. A FU cannot forward its result to waiting instructions and write its result to the destination register unless it gains access to the forwarding bus)

b. Drive the address of the destination register and the result on respective parts of the forwarding bus. The result also gets written to the destination register and also gets forwarded to instructions which are being renamed and dispatched.

- Instruction wakeup, selection and issue:

a. **Associative matching and data pickup:** As a result gets forwarded, IQ source register fields whose ready bits are not set and whose src register address field matches the address of the register driven on the forwarding bus do the following:

   – The register value driven on the forwarding bus is latched into the src register value field

   – The ready bit is set

   Note that if multiple forwarding buses are present, the src tag field in all src register fields that are not marked as ready have to be compared against the register address on each of the forwarding buses at the same time. This is because the result can be forwarded on any (and exactly) one of these buses

b. **Instruction wakeup:** IQ entries that have both sources marked as ready are eligible for issue. An instruction waiting for one or more source operands **wakes up** (= becomes ready for execution) when all of its sources are ready. Wakeups are thus a consequence of a match (or matches) in the course of forwarding. The wakeup logic associated with every allocated IQ entry simply ANDs the ready bits in the source fields of the IQ entry of that instruction. The instruction is considered to have awakened when this AND gate outputs a 1.

c. **Selecting ready instructions for issue:** Not all ready instructions can be issued simultaneously due to limitations on the connections between the IQ and the FUs and/or due to the existence of a finite number of FUs of a specific type. This step essentially **selects** a few of the ready instructions for actual issue as follows:

   - Ready IQ entries send a request for issue to the selection logic

   - The selection logic identifies instructions that can be issued

   - The operands for these selected instruction are read out from the IQ entries along with any necessary information into the appropriate FU to allow execution to commence in the *following* cycle.

   - The ready bits in the IQ entries vacated by issued instructions are cleared and these IQ entries are marked as free.

- **Timing requirement:** data forwarding, instruction wakeup, selection of instructions for issue and instruction issue are consecutive steps that all have to be completed within a single clock cycle. Equivalently,

$$T_{forward} + T_{match} + T_{wakeup} + T_{selection} + T_{issue} \leq T_{clock}$$

where:

$T_{forward}$ =	time taken to drive the result and the dest register address on the forwarding bus
T_{match} =	time needed to do a tag match
T_{wakeup} =	time needed to AND the ready bits of an IQ entry and drive the output of the gate as the request input to the selection logic
$T_{selection}$ =	time needed by the selection logic to select instruction to issue
T_{issue} =	time needed to move selected instruction to the appropriate FU.

Handling Dependencies over Condition Codes in Register Renaming

- Basic idea: CC architectural registers are handled just like the GP (general purpose) architectural registers.
- Hardware needed: ICC and FPCC registers (architectural); more physical registers (and associated W vectors, status fields etc.) to allow ICC or FPCC to be renamed.
- Rename table extended to add entries for ICC and FPCC
- When an instruction that requires flags in a CC as input is dispatched, the rename table is looked up to get the current stand–in physical register for that CC. (The required CC architectural register is treated just like an input GP architectural registers).
- When an instruction that sets ICC or FPCC is dispatched, a new *physical* registers is assigned to hold the CC values generated by this instruction.
- At time of completion, CC values and results are both forwarded on the result bus. Physical registers are deallocated as usual.

Acquiring Source Operands

- Dispatch-bound
  - Read PRF at time of dispatch
  - Requires source data fields in VFUs
- Issue-bound
  - Read PRF at time of issue
  - Makes VFUs narrower
  - Requires more ports on PRF
  - Delays freeing of physical registers

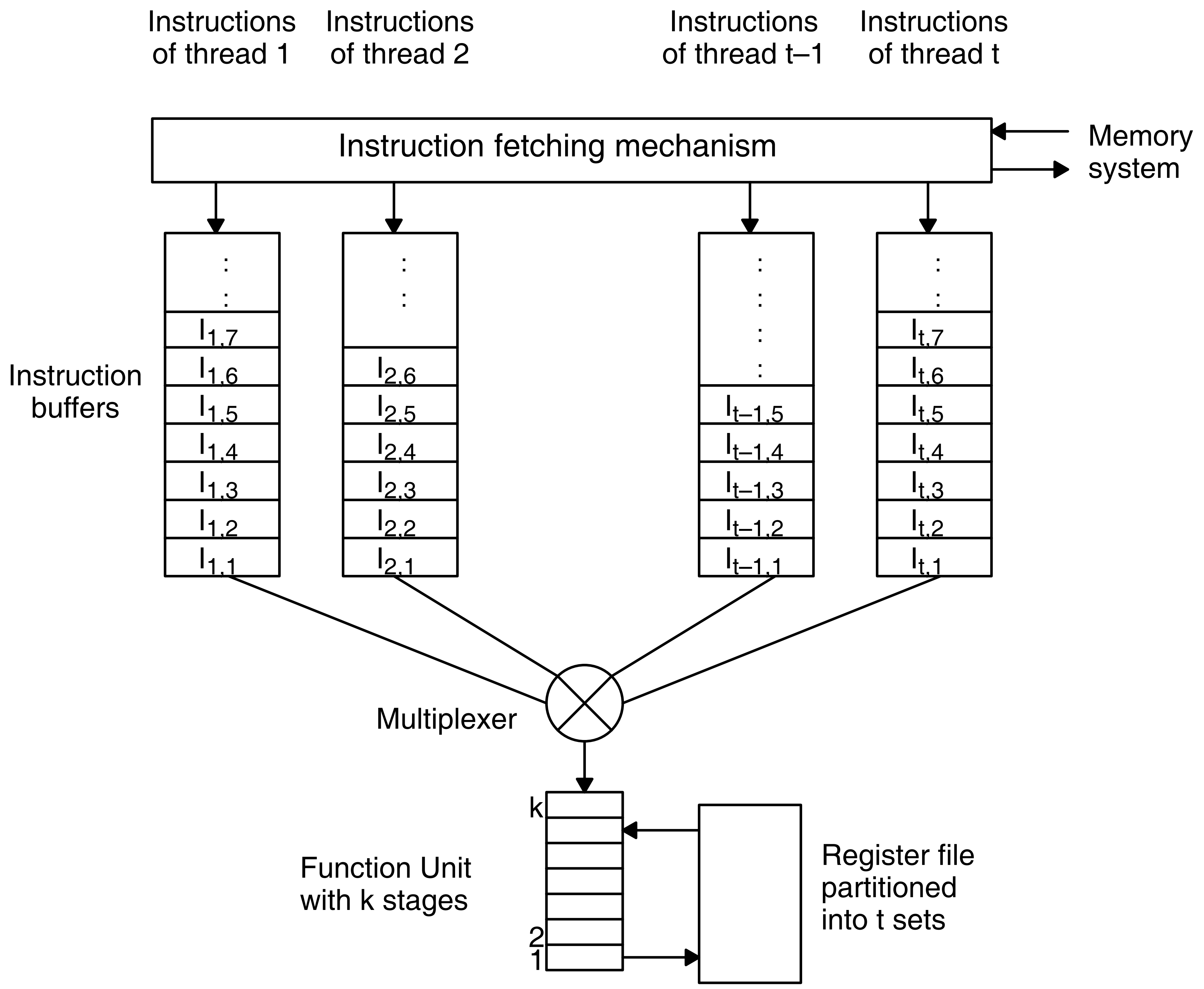
Simultaneous Multi-threading

- Independent threads executed on the same pipeline.
- Reduces or eliminates stalls due to data dependencies
- Examples:
    - Pentium 4, Sun Niagara, barrel processor

Characteristics of SMT

- One execution engine
- Shared by multiple threads
- Fill stalls in one thread with instructions from another.
- Good for out-of-order:
  - Memory stalls still a problem, limits to instruction-level parallelism
- Even better for in-order:
  - No concept of sequential execution model ***across*** threads

- Basic hardware setup is as shown:


Instructions of thread 1
Instructions of thread 2
Instructions of thread t–1
Instructions of thread t
Instruction fetching mechanism
Memory system
Instruction buffers
$I_{1,7}$ $I_{1,6}$ $I_{1,5}$ $I_{1,4}$ $I_{1,3}$ $I_{1,2}$ $I_{1,1}$
$I_{2,6}$ $I_{2,5}$ $I_{2,4}$ $I_{2,3}$ $I_{2,2}$ $I_{2,1}$
$I_{t-1,5}$ $I_{t-1,4}$ $I_{t-1,3}$ $I_{t-1,2}$ $I_{t-1,1}$
$I_{t,7}$ $I_{t,6}$ $I_{t,5}$ $I_{t,4}$ $I_{t,3}$ $I_{t,2}$ $I_{t,1}$
Multiplexer
k
2
1
Function Unit with k stages
Register file partitioned into t sets

- ► Interlocked instructions from the same thread are separated by instructions from the other threads: in the above example, as long as $k < t$, two instructions from the same thread cannot be in the FU at the same time.

Dispatch patterns

- Synchronous
  - round-robin
- Asynchronous
  - Switch threads on stalls

Pros and Cons

- Benefits
  - Reduces load latency
  - Reduces or eliminates branch penalty
  - Reduces dependency penalty
  - Higher logic utilization, lower energy, higher throughput
- Drawback
  - Higher latency

Multiple Threads, Multiple Contexts

- Context info:
  - PC, register set
  - Thread ID

Exceptions

- External
  - Interrupt from I/O device
- Internal
  - Divide by zero
  - Page fault

Sequential Execution

- Architectural state:
  - Architectural Register File
  - Program Counter, PSW
- Save state exactly at faulting instruction
  - Everything before, nothing after
- Context switch
- Restore state and continue

Challenges

- Out-of-order CPUs reorder instructions
  - Logical time vs. Physical time
- Later instructions may have altered the processor state before the faulting instruction.
  - How to save and restore that state?

Option: Save physical state

- Reprocessing of later instructions is a waste of energy.
- Expose physical state to OS
- Complicated and heavy-weight
- Added complexity generally not beneficial

Better option: Maintain precise state

- Update an image of architectural state in program order
- For each instruction added to precise state:
  - All prior instructions are completed
  - No instructions following have modified architectural state.
  - Effects are atomic

- If the precise state at the point of transferring control to the interrupt handling routine is saved, then resuming processing after the interrupt consists simply of:

a) Establishing the processor state to the saved precise state.

b) Transferring control to the instruction at the address *A* associated with the precise state.

Interrupts in Pipelined CPUs: Some Terminology

- **Architectural registers** – registers visible at the ISA level, including flag registers.
- **Architectural state** – same as precise state.
- **Instruction retirement** – the step of writing the result of an instruction that has not caused an exception to update the precise state. Sometimes also called **instruction commitment**.
  - This is the process of updating the architectural state
- **Precise Interrupts**: an interrupt (and its handling) mechanism relying on the resumption from a precise state which is maintained by the hardware or reconstructed by the software.

Interrupt-handling techniques

- Allow imprecise state
- Disallow certain kinds of interrupts
- Software manages precise state (barriers)
- Design hardware to always maintain precise state

Trivial precise state

- Avoid out-of-order completions
    - Uniform FU latencies
- Variation:
    - All FUs must raise exception at the same stage

Re-order Buffer

- Instructions enter queue (RoB) during dispatch
- Instructions complete out-of-order
- Results of completion written to RoB out-of-order
- Completed instructions retired in-order, updating precise architectural state

History buffer

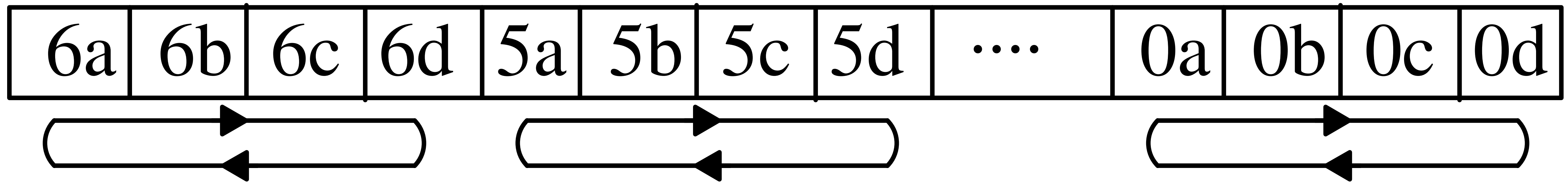
- Old register values are stored in history buffer as they are overwritten
- Old values tagged by overwriting instruction
- History buffer examined to reconstruct precise state -- Single-step in reverse program order

Future Files

- Two register files
  - Present updated in-order (on retirement)
  - Future updated out-of-order
- Interrupt serviced by swapping
- Must also swap metadata

Fine-grained register checkpointing

- Before updating a register, save old value to queue: Checkpoints that register value
- Old values can be stored in shadow registers:



- Reconstruction: Examine queue to identify which regs have to be restored to old values

Checkpointing Stack(s)

- Distributed: Each register is a stack.
- Centralized: One stack to hold all old values. [History Buffer]

Commit Barriers

- Instructions do not commit until barrier instruction is executed
- Barriers inserted by compiler
- Exceptions generated at commit

```
      :
FMUL F1, F5, F0
FADD F2, F1, F4
TRAPB
STORE R1, R6, #0
      :
```

Proper use of Barriers

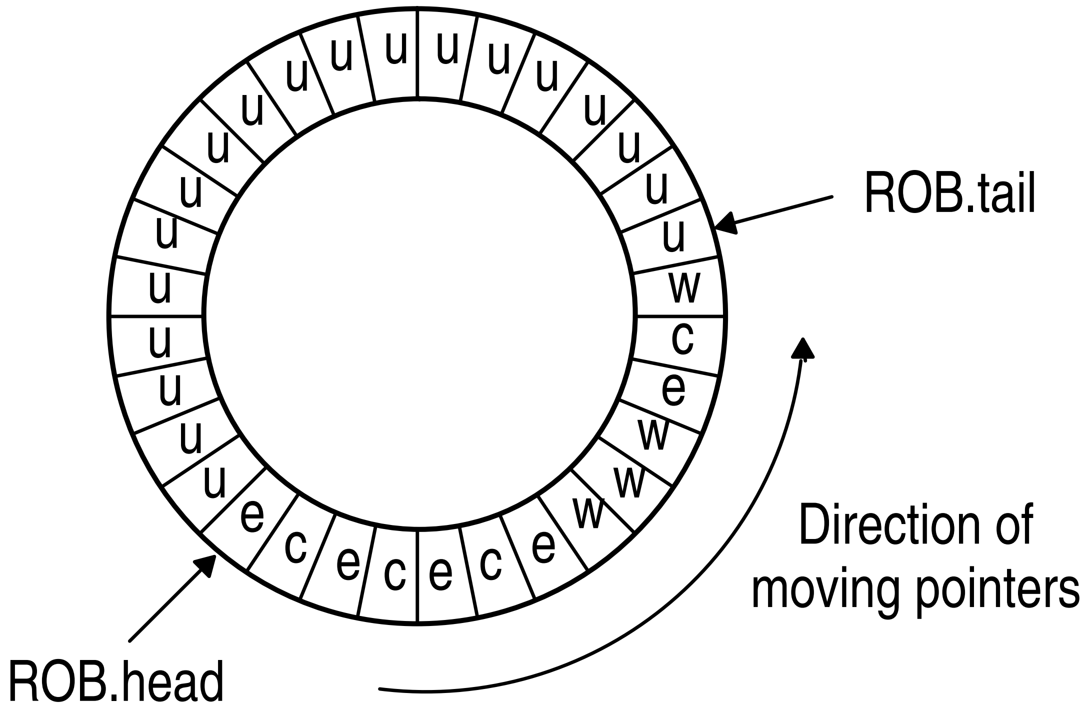
- Between barriers, each register assigned at most once (SSA)
- Only registers modified, no side-effects
- Insert barrier to catch exceptions
- Instructions after barrier trivially discarded
- Stores only after barrier

Tradeoffs

- More barriers:
  - Less work discarded on exception
  - Fewer registers tied up
  - More commit overhead
- Fewer barriers:
  - More work discarded on exception
  - When no exceptions: Faster, more out-of-order, less commit overhead

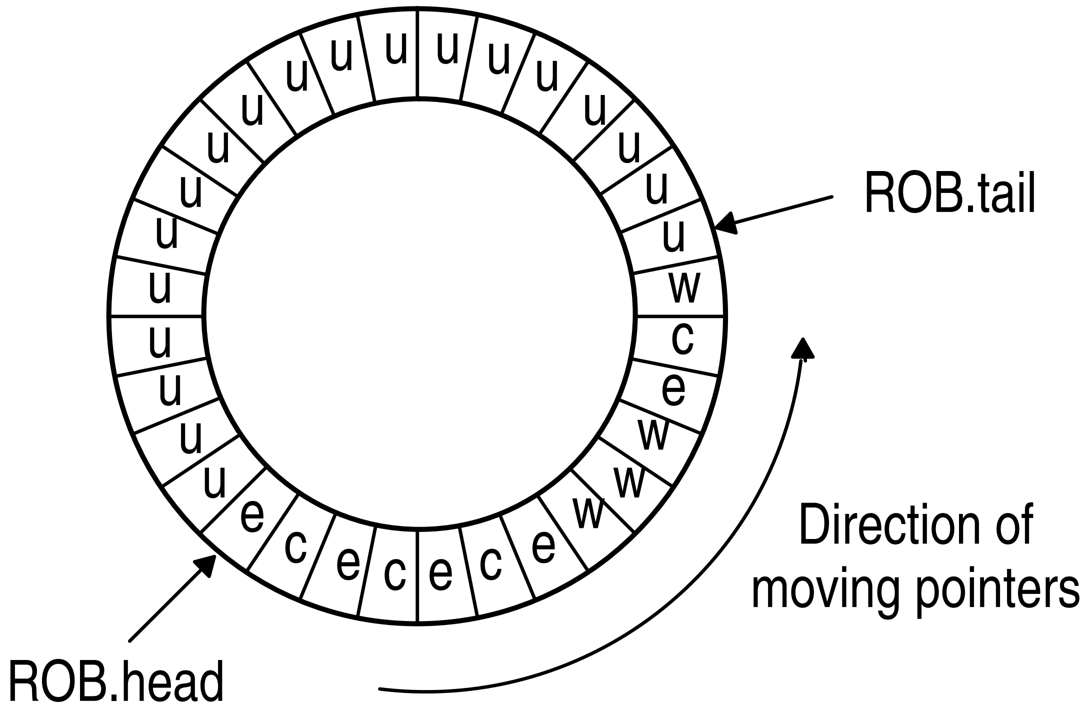
Reorder Buffers

- Dominant scheme for precise exceptions
- Complements register renaming
- Implemented as circular FIFO
- Common: RoB entries are the physical register file.

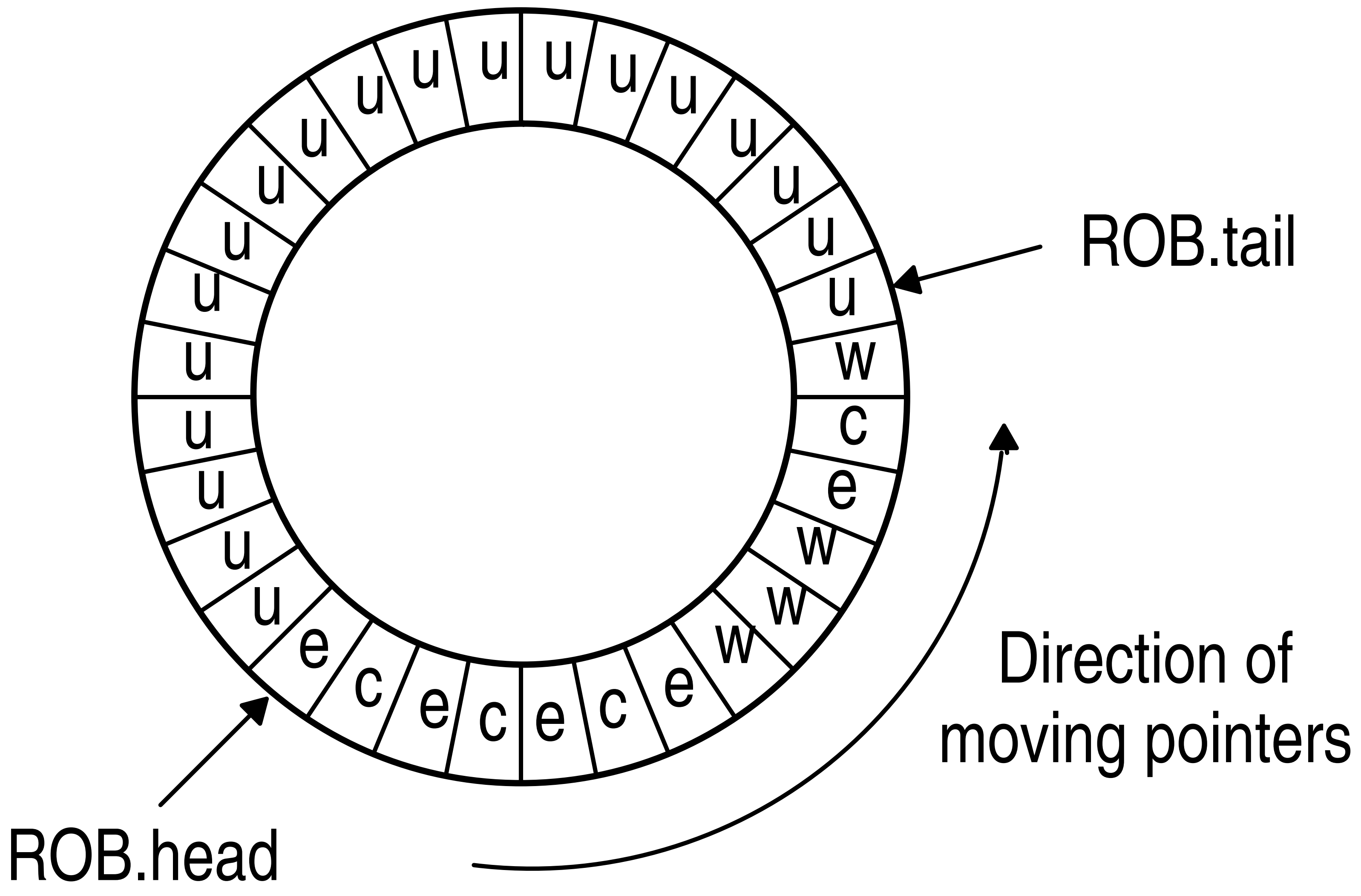


Dispatch

- Allocation:
  - Free space in queue?
  - Tail is new PRF entry, advance tail
  - Else: Stall



- The structure of the ROB:



e:	Instruction assigned to this slot is in execution
w:	Instruction assigned to this slot is waiting for input(s)
c:	Instruction assigned to this slot has completed
u:	This slot is unallocated

- The ROB is a circular FIFO, maintained in the order of instruction dispatch – that is, in program order.

Retirement

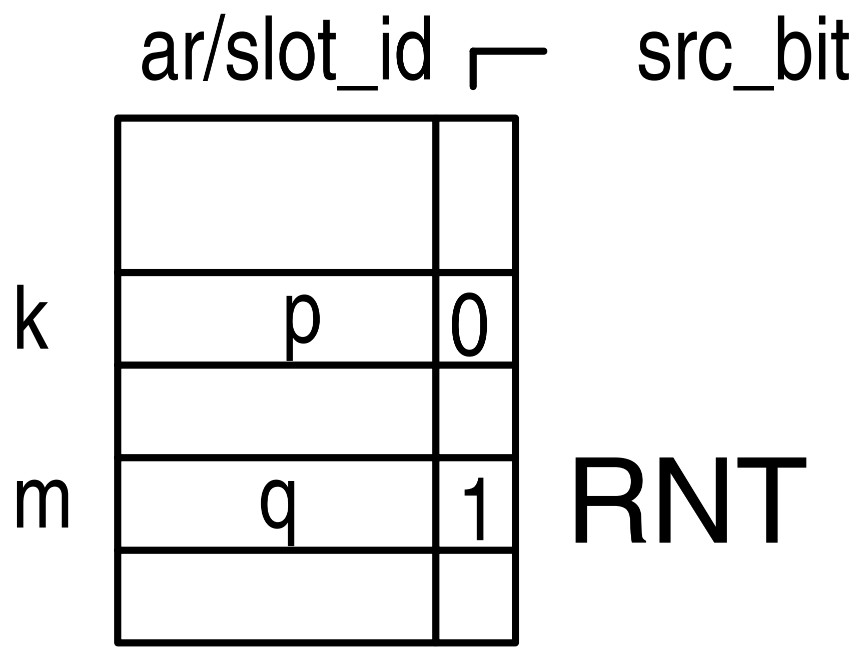
- The oldest completed instruction can retire
- External effects become visible
  - Exceptions are raised
  - Update to architectural state
  - Memory writes
- Dequeued by advancing head pointer

Return of the ARF

- Cannot leave retired instructions in RoB
- Commit results to ARF
- ARF holds precise architectural state wrt head of the RoB
- Corresponds to precise architectural state for oldest, retiring instruction

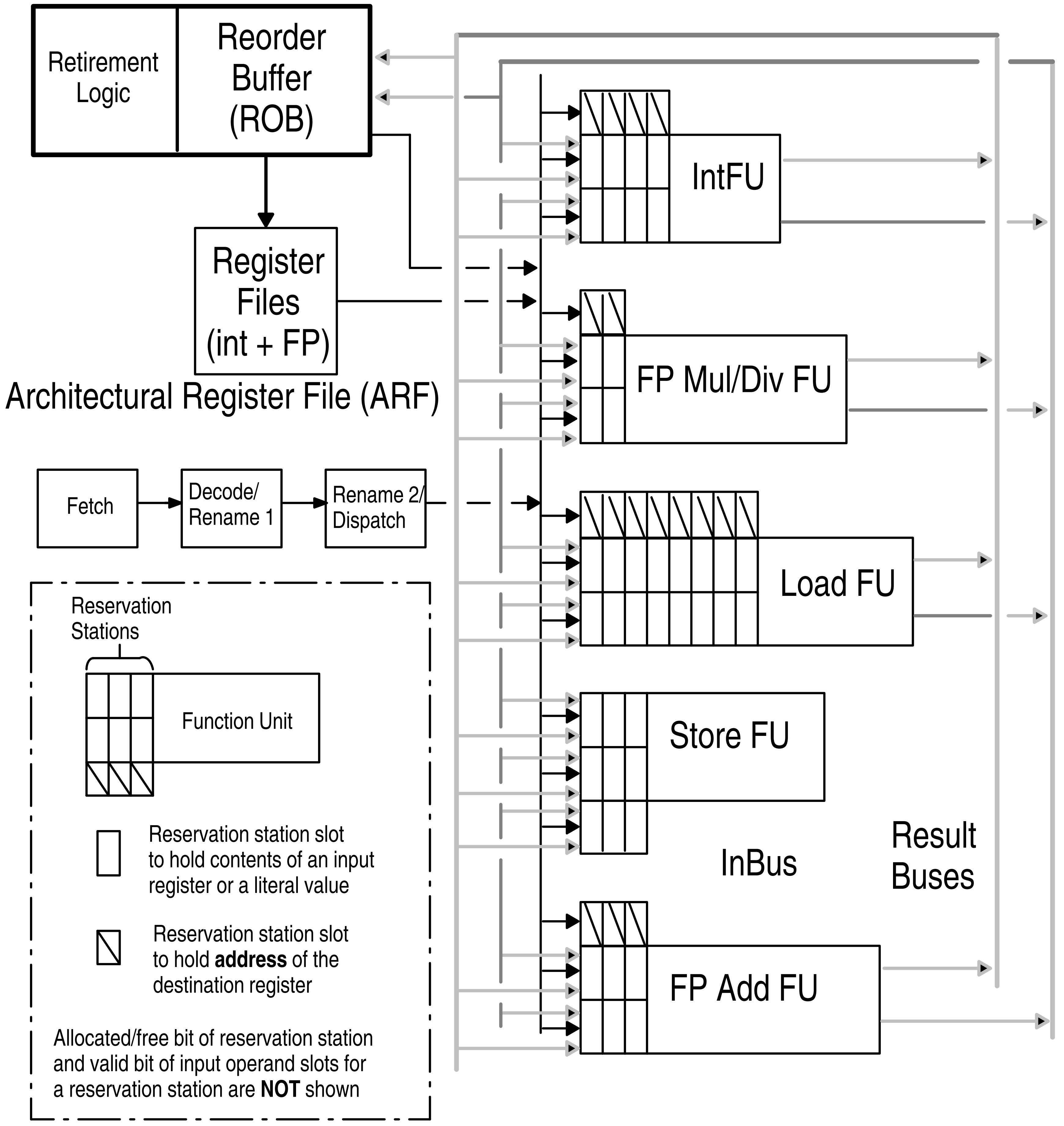
Modified Rename Table

- Rename table now points to RoB/PRF for uncommitted instructions
- Or ARF for committed results.



Alias for Rk is the p–th register in the ARF
Alias for Rm is the q–th slot in the ROB

- Datapath components:


Retirement Logic
Reorder Buffer (ROB)
Register Files (int + FP)
Architectural Register File (ARF)
Fetch
Decode/ Rename 1
Rename 2/ Dispatch
Reservation Stations
Function Unit
Reservation station slot to hold contents of an input register or a literal value
Reservation station slot to hold **address** of the destination register
Allocated/free bit of reservation station and valid bit of input operand slots for a reservation station are **NOT** shown
IntFU
FP Mul/Div FU
Load FU
Store FU
FP Add FU
InBus
Result Buses

Branch handling mechanisms are NOT shown

RoB entries

- Entry created at time of dispatch
- Contains:
  - Instruction address (PC)
  - Destination architectural register addr
  - Result value (for completion)
  - Exception codes
  - Status (e.g. result valid)

Allocate and Rename

- Require both free RoB entry and free VFU, else stall
- Use Alias table to translate sources:
  - If src_bit=0, look in ARF
  - else look in RoB/PRF

$\boxed{R_j \leftarrow R_k \ \mathbf{op}\ R_l}$

Instruction Processing

- Mark RoB entry allocated (advance tail) and filled out:

```
ROB[ROB.tail].status = invalid;
ROB[ROB.tail].PC_value = address of dispatched instruction;
ROB[ROB.tail].ar_address = j     (address of destination ar);
ROB[ROB.tail].excodes, result is left as it is;
Rename_Table[j].src_bit = 1   (latest value of Rj will be generated
                              within the ROB entry just set up);
Rename_Table[j].ar/slot_id = ROB.tail;
ROB.tail++
```

Forwarding

- Results are forwarded exactly as before
- RoB slot number == PRF index == the result tag
- Race between:
  - Writeback to PRF or commit to ARF
  - Dispatch

Completion

- Forwarded result written to RoB entry
- Modify RoB entry:

```
ROB[dest_slot_id].result = result produced by VFU
ROB[dest_slot_id].excodes = exception conditions produced
                            by the instruction during its
                            execution (can be all 0s, when
                            no excepetions have occurred)
ROB[dest_slot_id].status = valid
```

- RoB entry NOT deallocated

Retirement

- RoB.head points to next instruction to retire
- Stall if RoB is empty or head not valid
- if ROB[head].excodes:
  - Flush RoB of uncommitted entries
  - ARF, ROB[head].PC_valid contain precise state

Retirement

- If head valid and no exceptions, update architectural state:

```
ARF[ROB[ROB.head].ar_address]] = ROB[head].result;
if ( ((Rename_Table[ROB[head].ar_address].ar/slot_id
== ROB.head)
& (Rename_Table[ROB[head].ar_address].src_bit ==1) )
then {
    Rename_Table[ROB[head].ar_address].ar/slot_id =
            ROB[head].ar_address;
    Rename_Table[ROB[head].ar_address].src_bit = 0
};
ROB.head++
```

STOREs

- Enlarge ar_address to hold memory address.
- Flag to distinguish mem and ARF addresses
- Actual write only occurs at retirement
- Detect page faults at completion, raise exception at retirement.
- Pending writes also go to LSQ to service loads

LOADs

- Like reg-to-reg, LOAD instructions held in VFUs.
- Page fault detected early, sets valid and excodes in RoB, exception processed at retirement.
- On completion, read data written to PRF and satisfies dependencies.
- On retirement, read data written to ARF.

Handling other exception and other page faults:

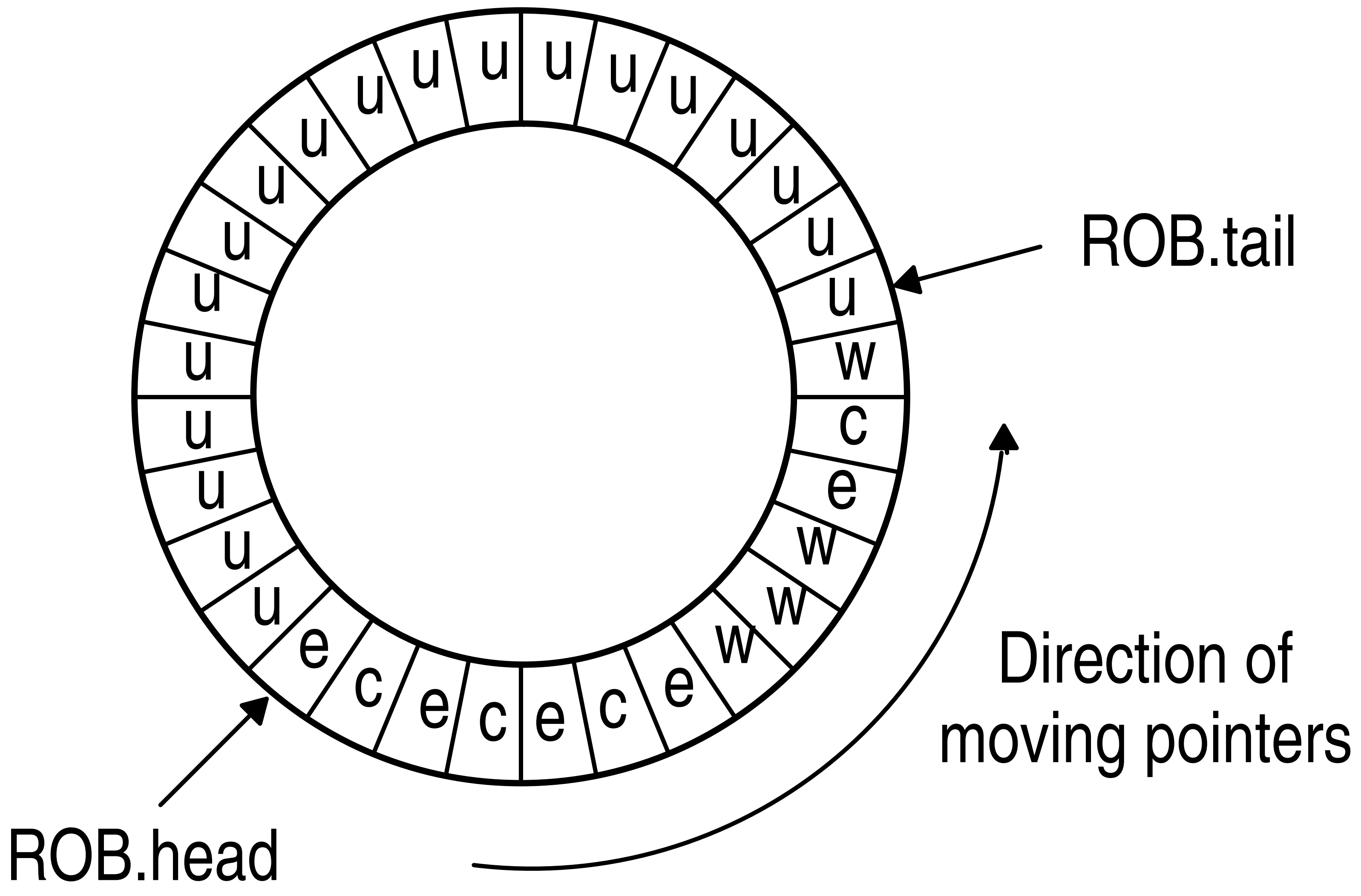
- ► Page faults triggered during instruction fetching, during the memory write of a STORE or by external sources are handled as follows:

– Page faults triggered by instruction fetch: further fetches are suspended and the pipeline is allowed to drain. If the retirement logic finds that a prior instruction generated an exception, the precise state is installed as before and the exception is serviced. If processing is to continue, the pending page fault induced by instruction fetching is also serviced.

Precise Exceptions Recap

- Exceptions: Unexpected change in program execution sequence
- May need to restart execution at faulting instruction
- Precise architectural state: Logical machine state of machine at instruction wrt sequential execution model.
- Challenge: OOO processor executes instructions out of program order:
    - No physical state corresponding to logical state
    - How to compute precise state to restore later?

- The structure of the ROB:



e:	Instruction assigned to this slot is in execution
w:	Instruction assigned to this slot is waiting for input(s)
c:	Instruction assigned to this slot has completed
u:	This slot is unallocated

- The ROB is a circular FIFO, maintained in the order of instruction dispatch – that is, in program order.

Retirement Recap

- Oldest complete instruction committed in program order:
  - Update precise architectural state (commit writes to architectural register file)
  - Process exceptions
  - Writes to memory
- ARF is image of precise architectural state

Real Examples:

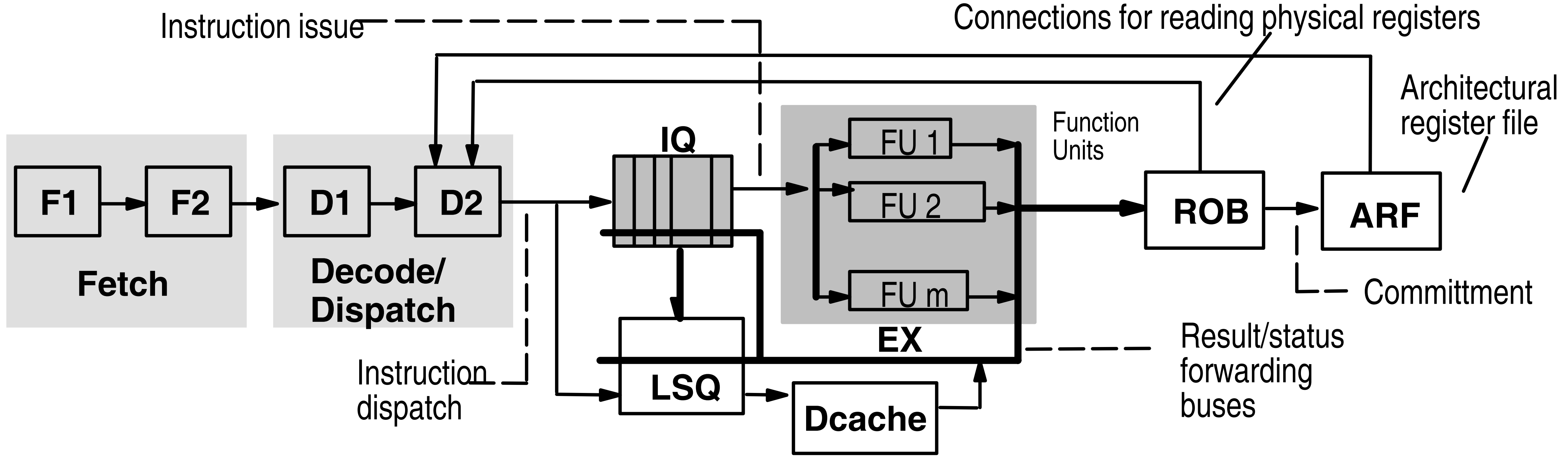
- Pentium P6 Pro: 40 entry ROB
- AMD K5: 16 entry ROB
- HP PA 8000: 56 entry ROB (the largest!) – occupies 15% of the die area

More recent RoB sizes

PowerPC 970	~200
Lynnfield (Nehalem)	128
Sandy Bridge	168
Ivy Bridge	168
Haswell	192

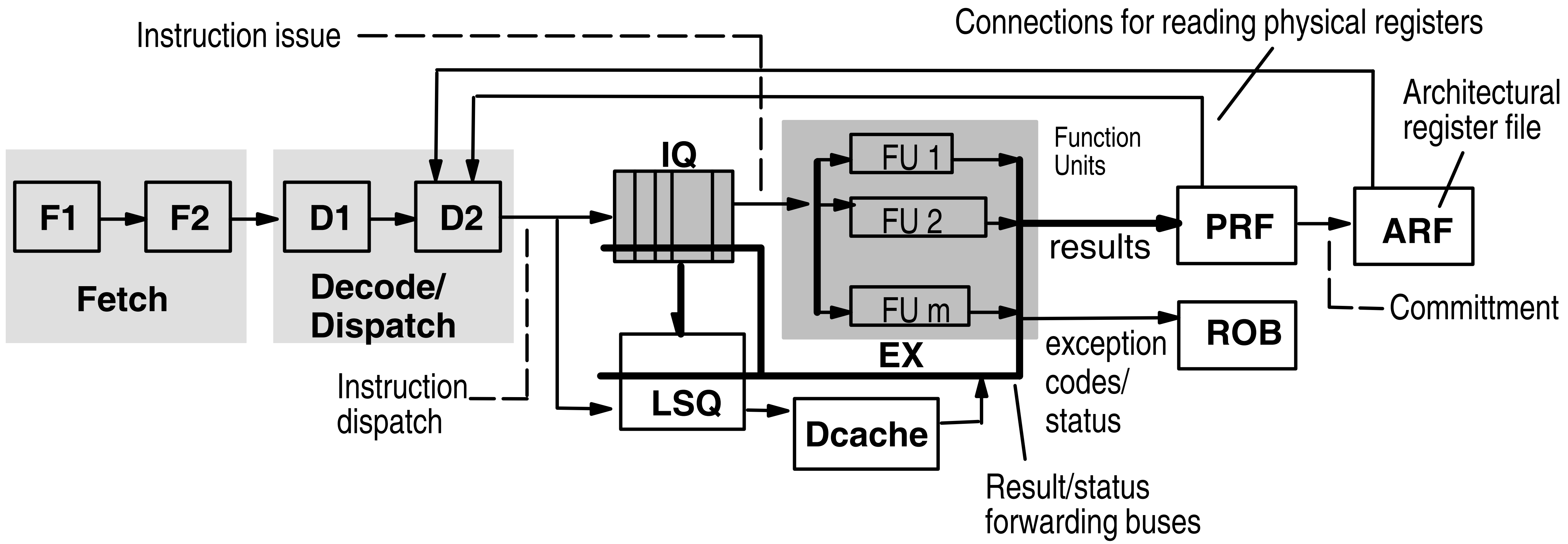
Real Machines: Implementations of Register Renaming

- **Variation 1:** Reorder buffer slots are used as physical registers and a separate architectural register file (ARF) is used (what we have just seen). The Intel P6 microarchitecture, implemented by the **Pentium Pro, Pentium, Pentium II and Pentium III** are examples of this design.



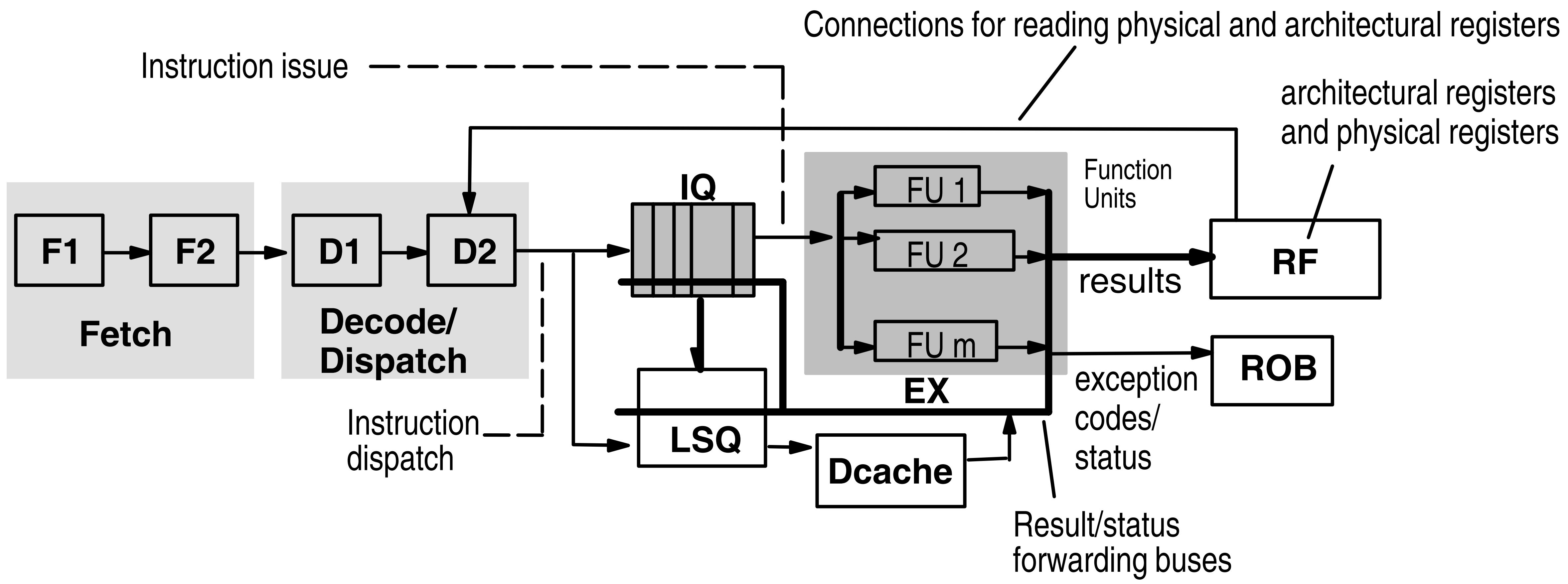
► The **LSQ** (load–store queue) shown is used to maintain the program–order of load and store instructions. Memory operations have to be performed in memory order. This ordering can be relaxed only when the address of a later load in the LSQ does not match the address of every store in front of it in the LSQ (LATER)

- **Variation 2:** A separate physical register file (PRF) and an architectural register file is used. The ROB entry of an instruction that has a destination register points to the physical register is assigned to hold the result. In this case, the physical registers are called **rename buffers**. The **IBM Power PC 604** implements this design.



► The physical registers can be allocated in a circular FIFO fashion (like ROB entries) or they can be allocated using a list or lists to keep track of free and allocated registers.

- **Variation 3:** The physical and architectural registers are implemented within a common register file. The **Intel Pentium 4** and the Alpha 21264 implementation uses this design.



Unified register file

- Uncommitted RAT:
  - Updated on dispatch — new instructions
- Retirement RAT (RetRAT):
  - Updated on retirement
- On retirement, no need to move data, just update RetRAT
- Complicates PRF (de)allocation

Superscalar Processors

- Dispatch more than one instruction per cycle
- Alternative to deeper pipelining and faster clock
- Keep execution logic busier

http://coeunitedparallelprocessing.blogspot.com/2014/03/superpipeline-and-superscalar.html

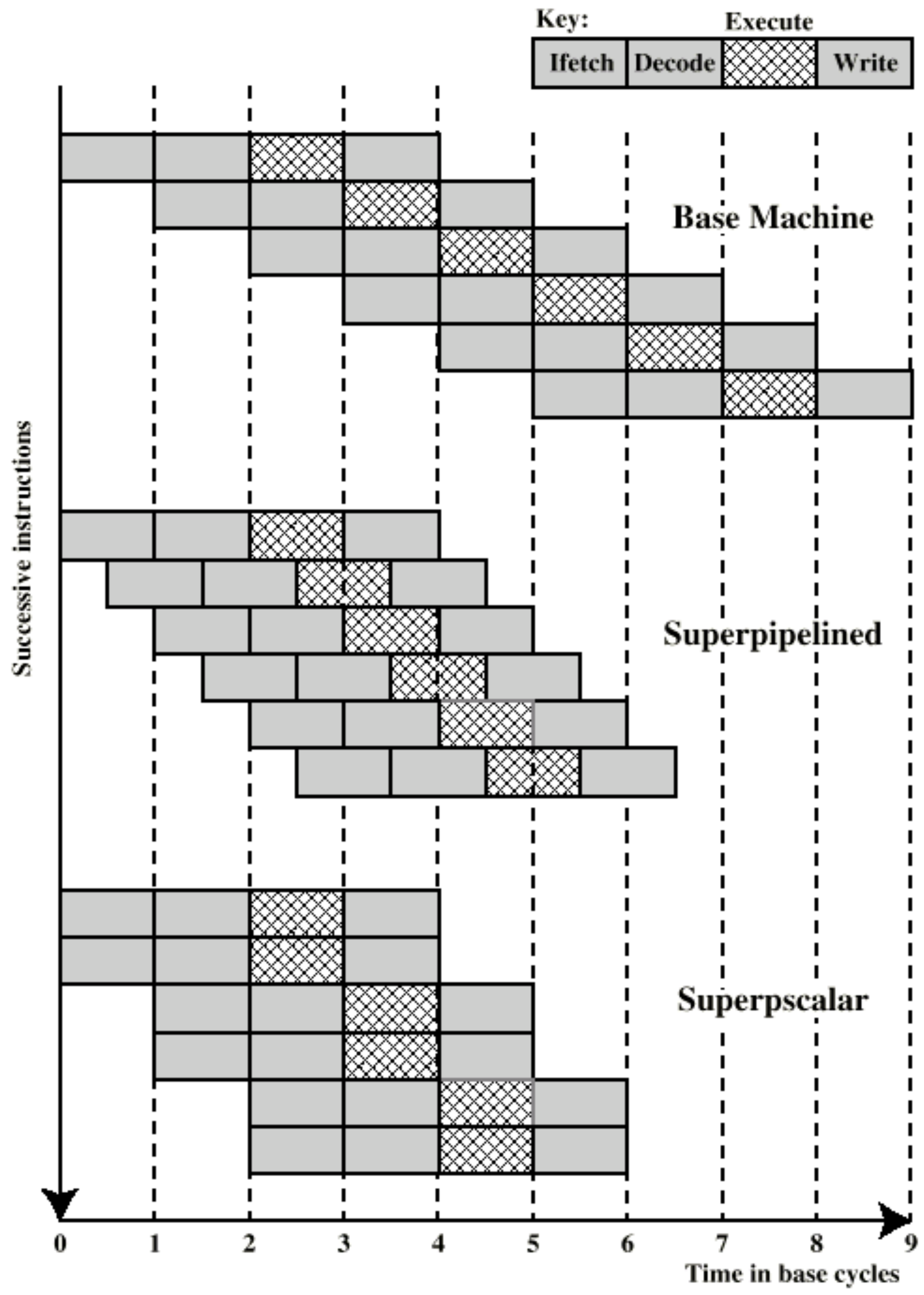


Figure 13.2 Comparison of Superscalar and Superpipeline Approaches

Execution time

$$T_{exec} = N \times \tau \times CPI$$

- N = Instructions executed
- τ = Clock period
- CPI = Average clocks per instruction

Underutilization

- `I0: ADD R1,R2,R4`
- `I1: MUL R4,R5,R6`
- `I2: ADD R7,R8,R9`
- `I3: MUL R10,R11,R12`

Motivation

- Max completion rate == Max dispatch rate
- Dispatch is bottleneck
- Underutilized functional-unit and instruction-level parallelism

Superscalar

- Fetch and Dispatch more than one instruction per cycle (m>1)
- Complex Fetch and Dispatch logic
- Complex dependency resolution

Benefits

- Because of FU parallelism and ILP:
  - Potentially CPI < 1
- Higher logic utilization is more energy-efficient

Drawbacks

- Logic complexity
- Increased cycle time (tradeoff)
- Limits to ILP
- Superlinear power and area

Relative Drawbacks

- Relative to maximum dispatch rate:
  - Increased branch mispredict penalty
  - Increased memory latency

Multiple Dispatch

- “m-way” superscalar
- Maximum of m instructions dispatched per cycle
- Ideal CPI = 1/m
- Common widths
  - m=2 -- Early and low-end modern
  - m=4 -- Common with modern high-end
  - 6 to 8 -- Very aggressive

Diminishing Returns

- Actual CPI does not go down by 1/m.
- Sequential execution model: Dependencies
- Branches
  - Resolution of branch direction
  - Mispredict penalty
  - Multiple branches in same group

Other challenges

- Block of m instructions may cross memory boundary (cache line, page)
- Dependency resolution:
  - Instructions dispatched on earlier cycles
  - Dependencies among instructions dispatched together

What doesn’t change

- Instruction issue
  - But instruction queues fill faster
  - Might demand duplicate FUs
- Multiple retirement
  - Have to drain RoB as fast as it can be filled

Other ramifications

- Wider internal buses:
  - Width(Fetch) ≥ Width(Decode) ≥ Number of dispatches per cycle
- Reservation stations must match issue rate to dispatch rate:
  - May require duplicate FUs
- Faster, more complex memory interfaces, register file, reorder buffer:
  - Wider buses, more ports

Superscalar Dispatch

- Resources available & instructions independent: Dispatch all m at once.
- Dispatch instructions in program order, stopping at first that cannot be dispatched.

Checking Dependencies

`I1, I2, I3, ... Im`

- Resolve dependencies `I1` to `Im` in program order
- Resolve dependencies with earlier dispatched instructions
- With renaming, only resolve flow dependencies

Complexity of checking

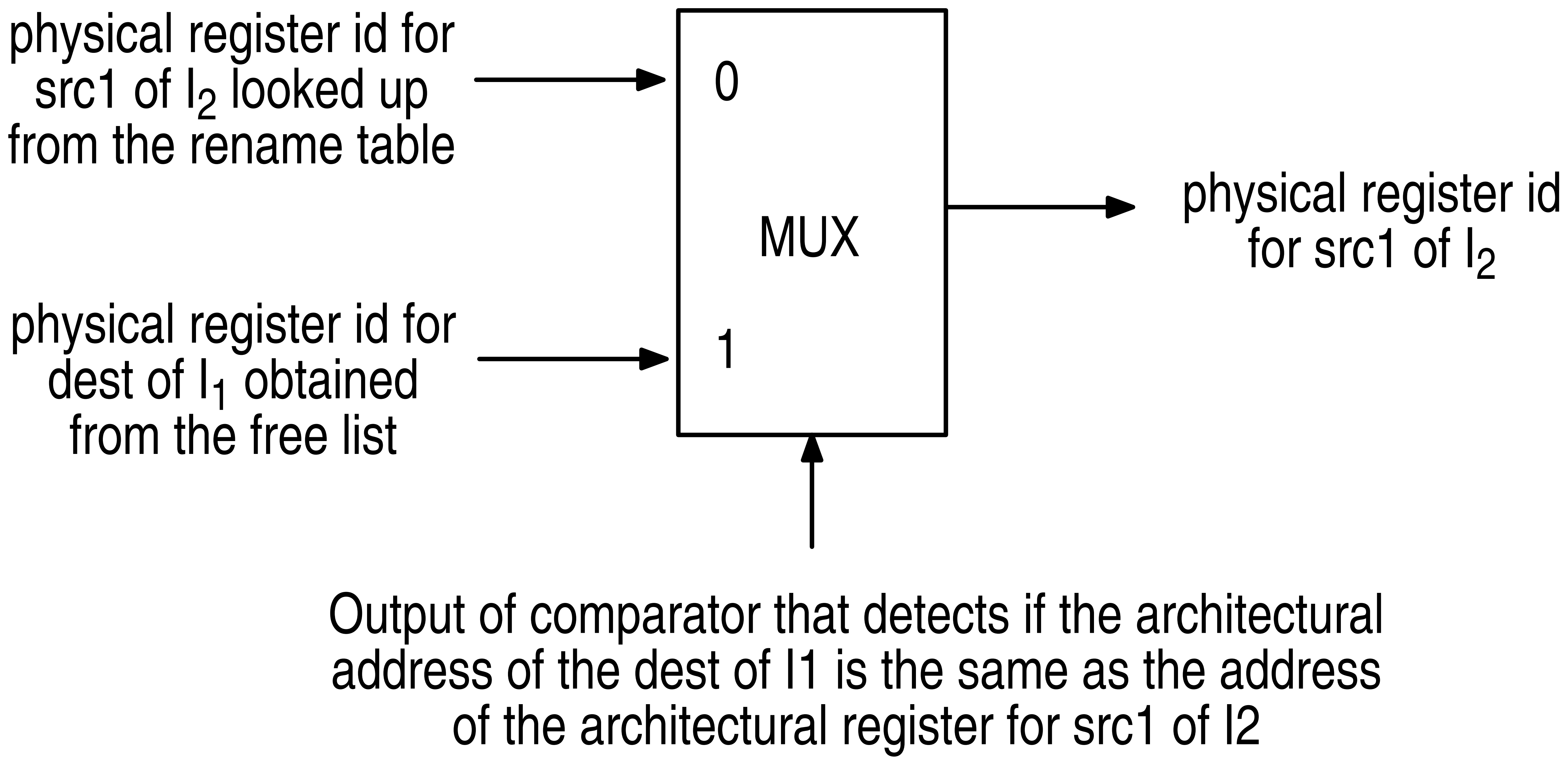
```
I1, I2, I3, ... Im
```

- Assume 2 sources, 1 dest
- `I1` ➔ `I2, I3, ... Im`
  - 2 * (m-1) comparators
- `I2` ➔ `I3, ... Im`
  - 2 * (m-2) comparators
- m * (m-1) is O(m^2)

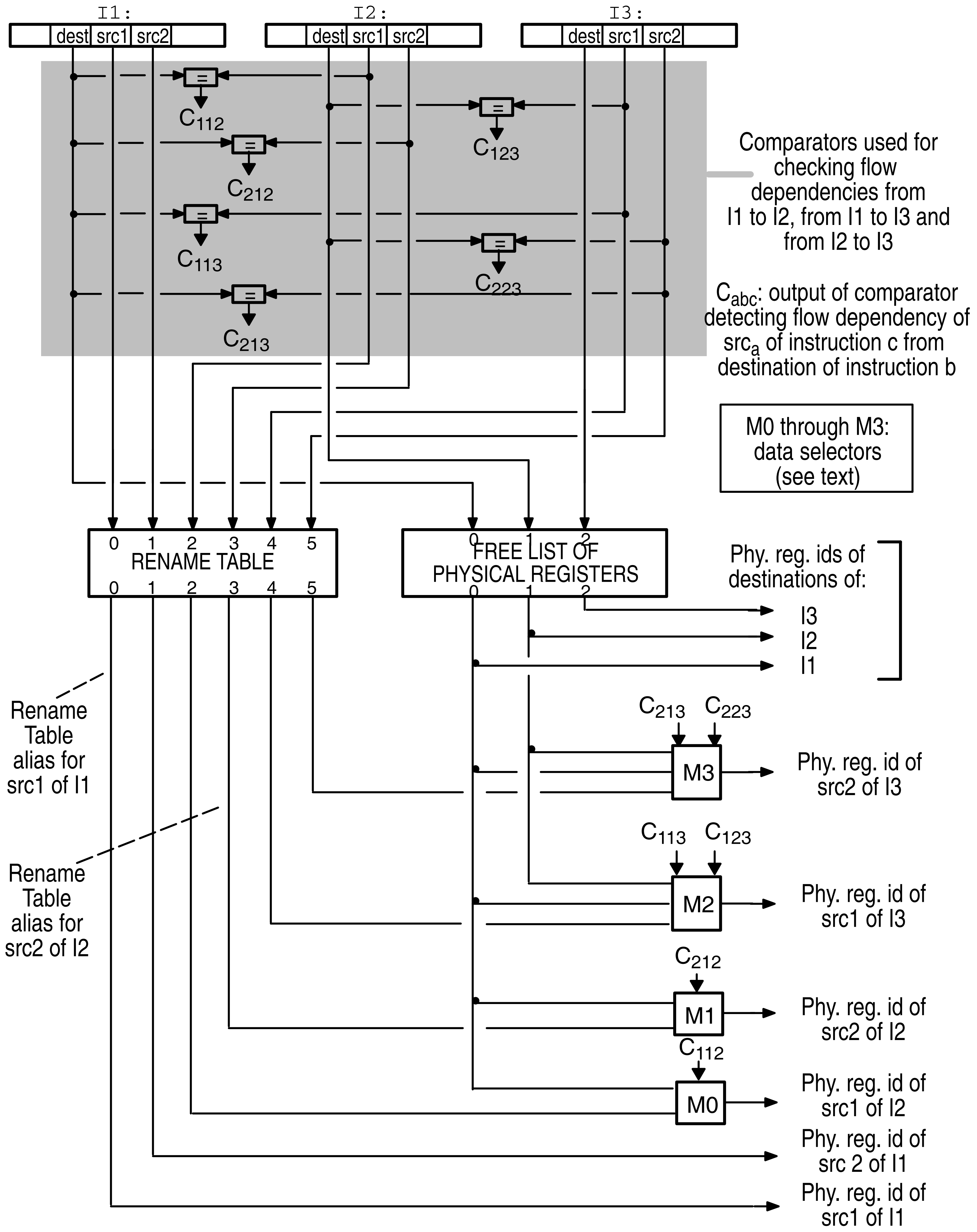
Assignment of Physical Registers

- Sources for `I1` found in rename table
- Sources for `I2`:
  - Look in rename table
  - Compare to `I1` dest
    - Physical register assigned to `I1` dest may be `I2` source

Physical Register Source for I2



physical register id for src1 of I_2 looked up from the rename table
0
MUX
physical register id for src1 of I_2
physical register id for dest of I_1 obtained from the free list
1
Output of comparator that detects if the architectural address of the dest of I1 is the same as the address of the architectural register for src1 of I2


I1:
dest src1 src2
I2:
dest src1 src2
I3:
dest src1 src2
C112
C123
C212
C113
C223
C213
Comparators used for checking flow dependencies from I1 to I2, from I1 to I3 and from I2 to I3
Cabc: output of comparator detecting flow dependency of srca of instruction c from destination of instruction b
M0 through M3: data selectors (see text)
RENAME TABLE
FREE LIST OF PHYSICAL REGISTERS
Phy. reg. ids of destinations of:
I3
I2
I1
Rename Table alias for src1 of I1
Rename Table alias for src2 of I2
M3
Phy. reg. id of src2 of I3
M2
Phy. reg. id of src1 of I3
M1
Phy. reg. id of src2 of I2
M0
Phy. reg. id of src1 of I2
Phy. reg. id of src 2 of I1
Phy. reg. id of src1 of I1

Supporting Back–to–Back Execution

- Consider the following instruction sequence:

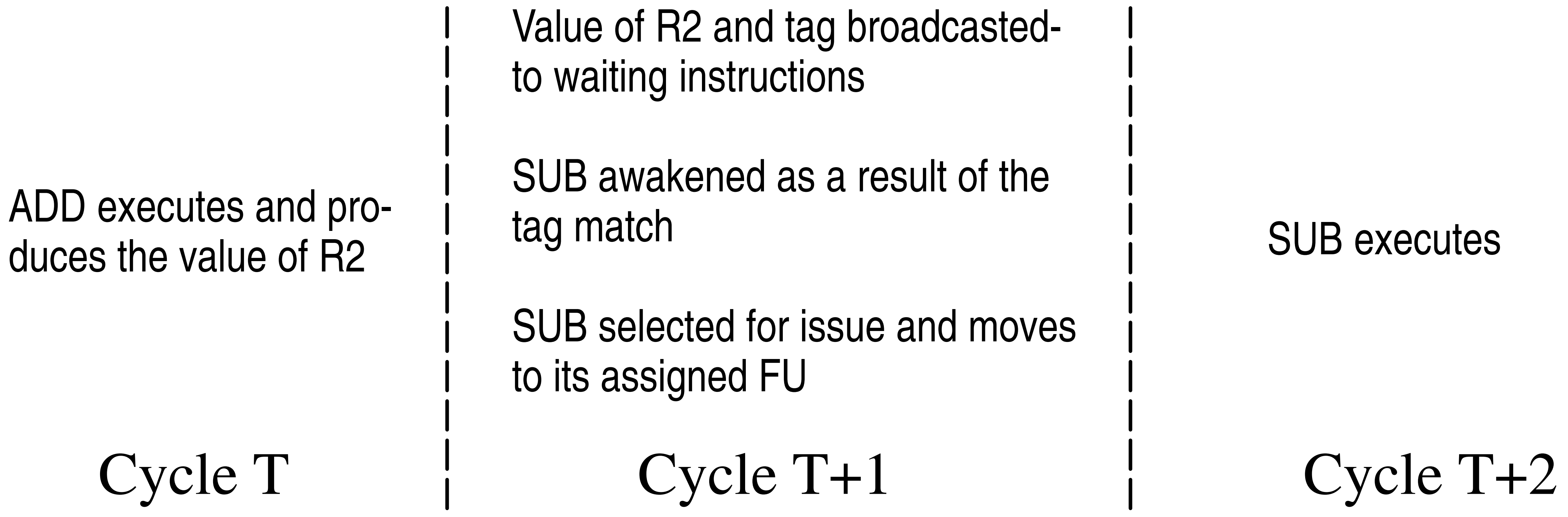
```
ADD  R2, R4, R5
SUB  R6, R2, #4
```

  In this code sequence, if both the ADD and the SUB have been dispatched, the SUB has to wait for the value of R2 to be produced before it can start up.

- ► Ideally, the SUB must issue as soon as the value of R2 is available. The ADD and the SUB are then said to ***execute back–to–back***

- ► With the use of an IQ, such a back–to–back execution is only possible if the value of R2 is forwarded to the SUB as it moves to the assigned FU, after being issued.

- This, in turn, implies that the SUB must be selected for issue one cycle **before** the ADD completes execution, so that it can be awakened and selected in the cycle before, permitting the SUB to move out to its assigned FU as the result of the ADD is forwarded to waiting instructions. Unfortunately, this is what happens in the wakeup logic discussed on Pages 160–164:


ADD executes and produces the value of R2
Value of R2 and tag broadcasted to waiting instructions
SUB awakened as a result of the tag match
SUB selected for issue and moves to its assigned FU
SUB executes
Cycle T
Cycle T+1
Cycle T+2

There is a one cycle between the startup of the executions of the ADD and SUB – that is, back–to–back execution is not supported.

- Back–to–back execution is supported if the tag of R2 is broadcasted one cycle before the value of R2 is available:

ADD executes and produces the value of R2 Tag of R2 broadcasted to waiting instructions SUB selected for issue and moves to its assigned FU	Value of R2 picked up by waiting instructions in the IQ that had a match with the tag broadcasted in the **previous** cycle SUB receives the value of R2 directly from the forwarding bus to the input of the assigned FU SUB executes
Cycle T	Cycle T+1

Back-to-back dependencies

- Split forwarding bus:
  - Cycle n-1: Broadcast tag
  - Cycle n: Broadcast data
- Dependent instructions wake one cycle early and take source data from bus