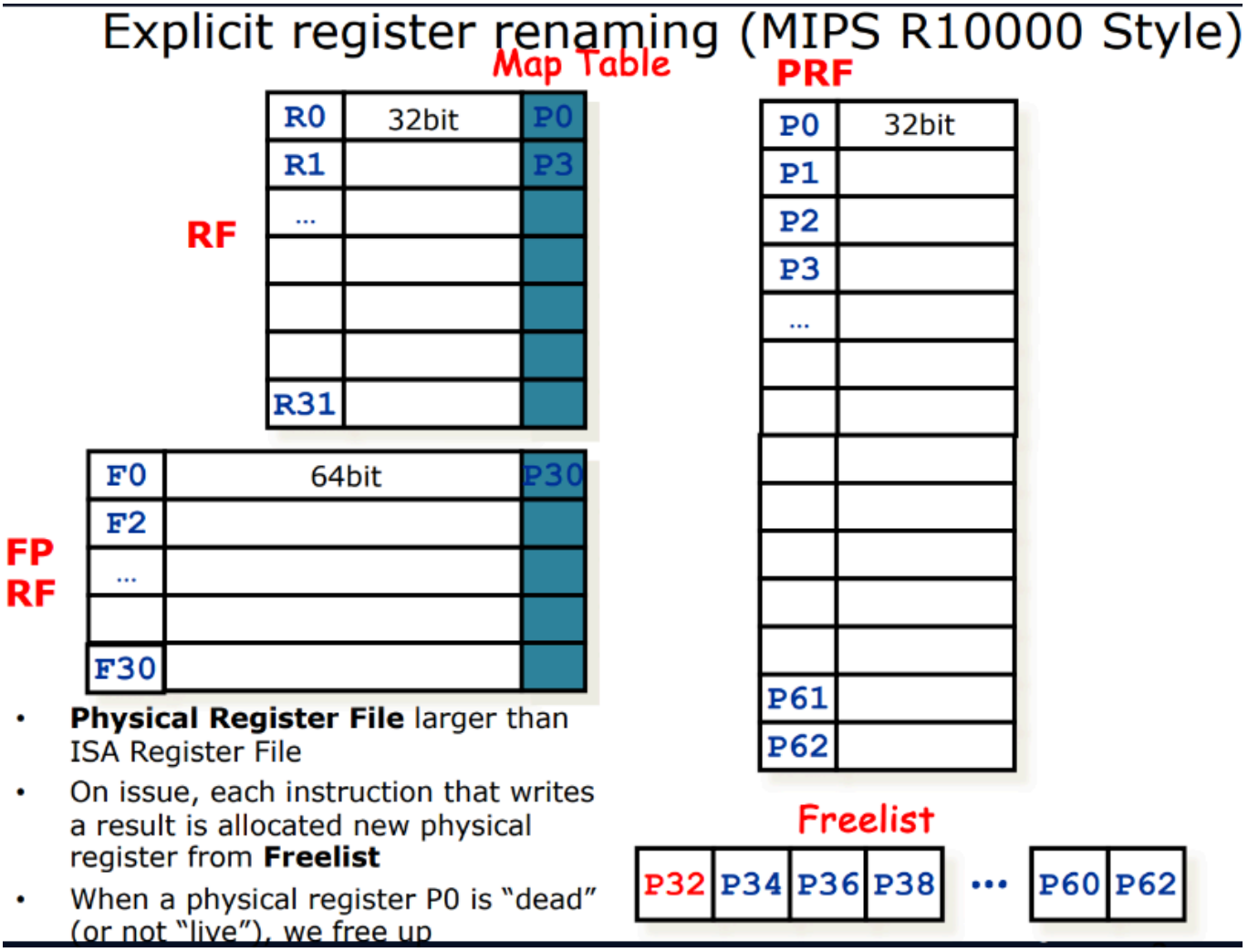
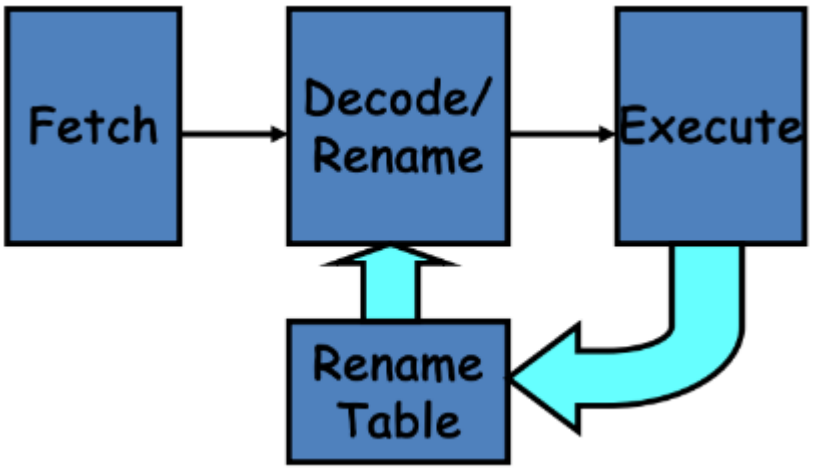
10.Explicit Register Renaming

Tomasulo provides Implicit Register Renaming and it is used to move the value in the reservation space.
Now we introduce the Explicit Register Renaming:

- we use physical register file that is larger than the number of registers specified by the ISA(Example: form MIPS is 32 registers, we have the freedom to use another register than the one I am using, it works well when the registers are more than the one in the ISA)
  The key idea is that the physical register is determined when we do a write to a register(Not only when we write for the first time in a register). When we write in the register it becomes the new physical register available. If we write to a new one I have to be sure I am not overwriting some data that are still relevant. Similar to the concept of SSA in compiler(Associate an index to the value). Every time we write in a register we redefine a physical one that will be referred to the operations(No more WaW and other WaR as the two registers are separated).
  This is useful to do efficiently out of order operations. Kind of dynamic compilation.



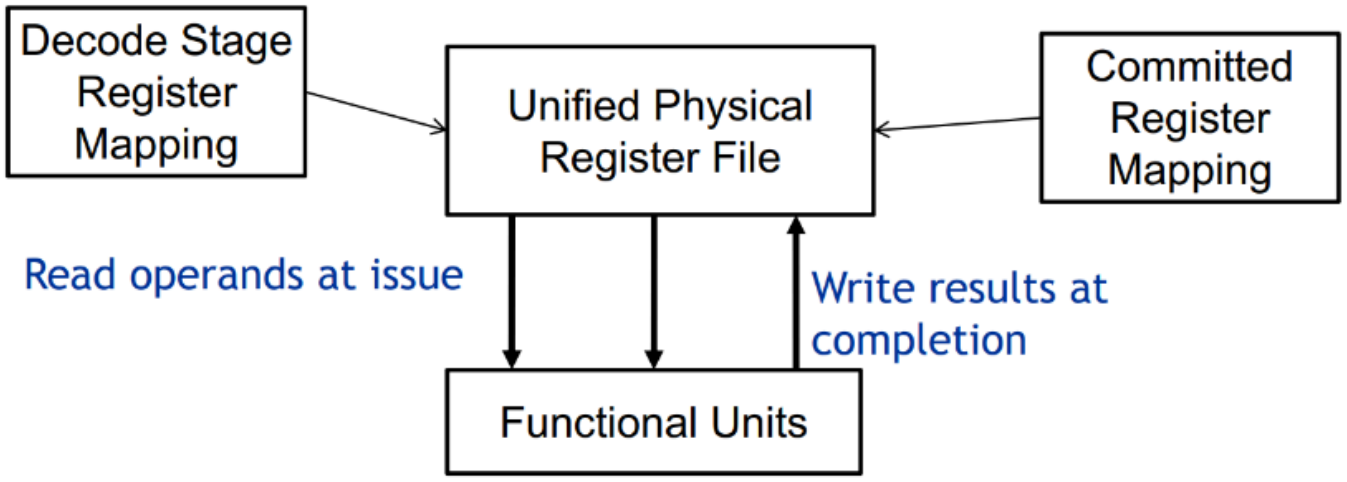
Mechanism



You have to keep a translation table:

- ISA register → physical register mapping
- When register written, replace entry with new register from freelist
- Physical register becomes free when not used by any active instructions

Unified Physical Register File

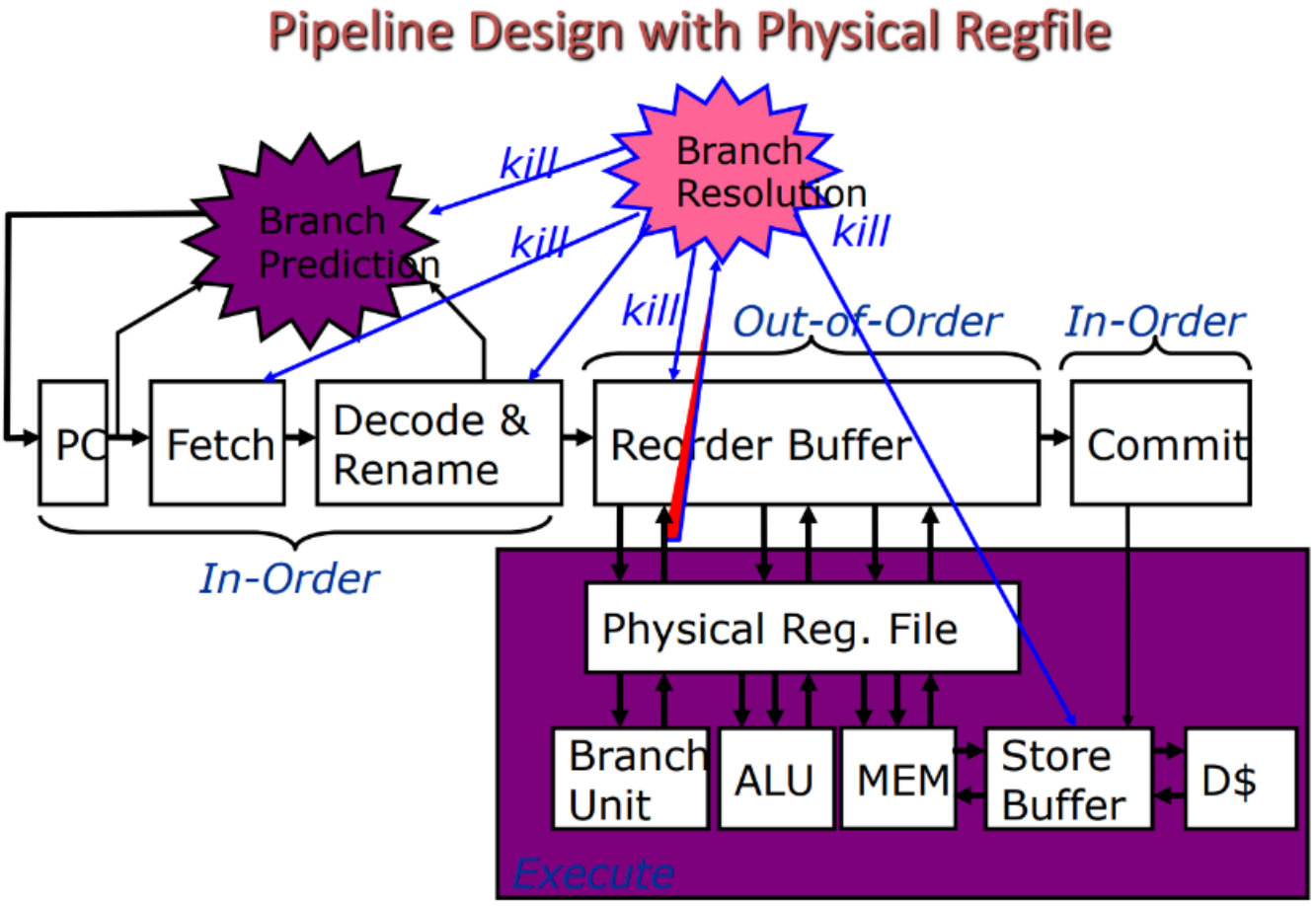


Rename all architectural registers into a single physical register file during decode, no register values read.
Functional units read and write from single unified register file holding committed and temporary registers in execution
Commit only updates mapping of architectural register to physical register, no data movement

Instruction Commit

Record that the mapping between an architectural register number(number of the register in the architecture)and physical register number is no longer speculative, in the sense that we now have to define the new physical register.
If it is the last use of the architectural register you can free all the previous physical register used to hold the older value of the architectural register.
Physical register could have many uses and they could be operand of operation in queue. If

there are no outstanding use we can deallocate the physical register. If there are outstanding uses I don't deallocate the register and go on with the operation and continue so until the register is no longer used.
For doing so we need a data structure to provide the physical register corresponding to the architectural register. Then, during the commit, I update this table to say that a physical register containing the destination value corresponds to that architectural register.



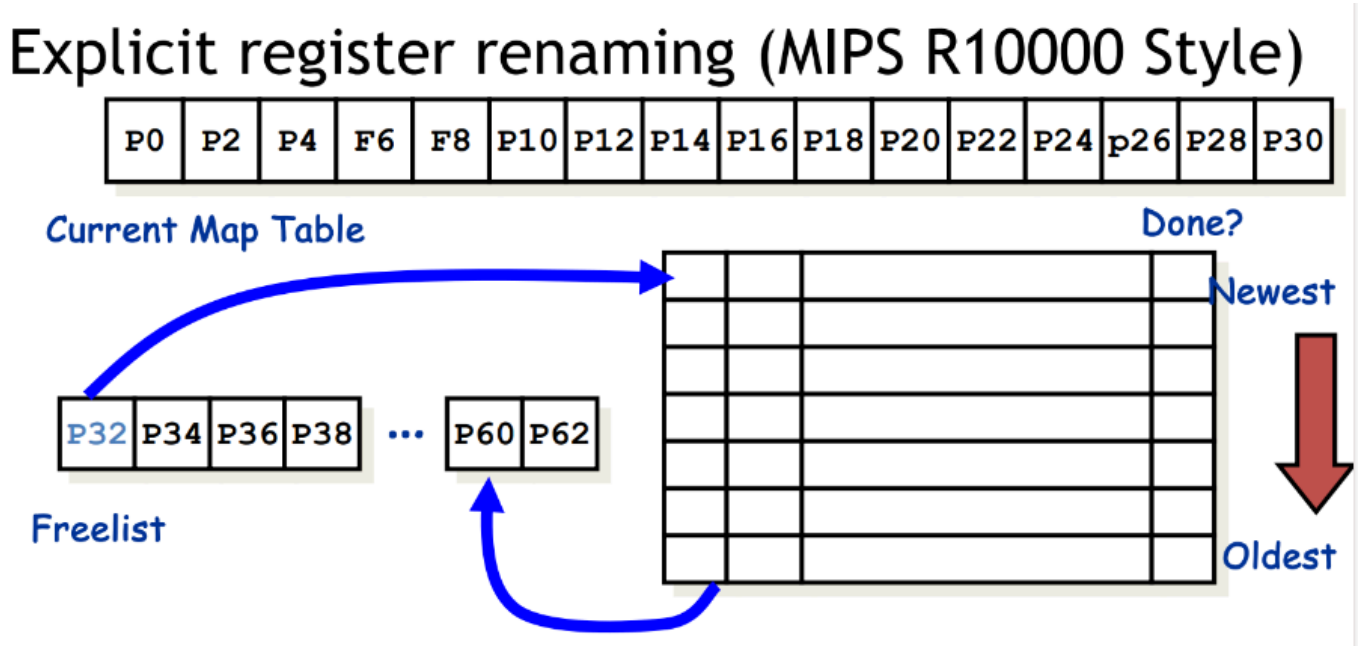
If you have a misprediction you may need to flush the outstanding instructions(not commit the instruction until the prediction is done).
Decouples renaming from scheduling:

- Pipeline can be exactly like “standard” DLX pipeline (perhaps with multiple operations issued per cycle)
- Or, pipeline could be tomasulo-like or a scoreboard, etc.
- Standard forwarding or bypassing could be used
  Allows data to be fetched from single register file
- No need to bypass values from reorder buffer. Important as it come back in standard pipeline
- This can be important for balancing pipeline
  Many processors use a variant of this technique:
- R10000, Alpha 21264, HP PA8000
  You need to have a way to check which register are free, otherwise you cannot execute an instruction as there couldn't be space to save the results.

The efficiency is that we can start the execution of multiple instructions at the same time. We need to understand how to update the free list and how it is combined with the precise interrupts.

First we need to analyze the lifetime of the physical register. We need to see if the register contains real values or speculative data related to speculative execution. The lifetime of a physical register is the time between the first write and the last use of the value in the register.

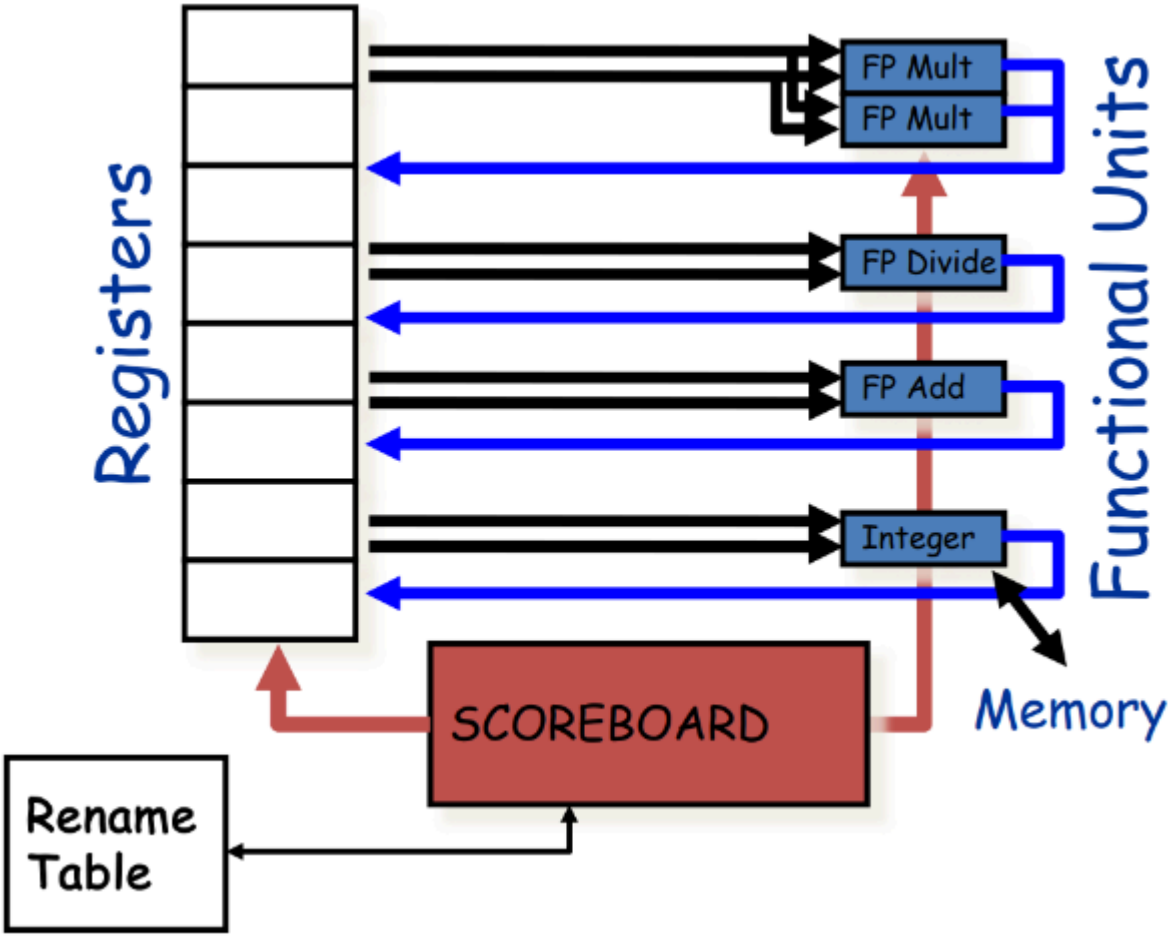
We will use a map table to map values and instructions. On issue each instruction that modifies a register is allocated new physical register from freelist



Makes speculative execution/precise interrupts easier:

- All that needs to be “undone” for precise break point is to undo the table mappings

Explicit Register Renaming with Scoreboard



Issue—decode instructions & check for structural hazards & allocate new physical register for result

- Instructions issued in program order (for hazard checking)
- Don’t issue if no free physical registers
- Don’t issue if structural hazard
  Read operands—wait until no hazards, read operands
- All real dependencies (RAW hazards) resolved in this stage, since we wait for instructions to write back data.
  Execution—operate on operands
- The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard
  Write result —finish execution
  Note: No checks for WaR or WaW hazards!
  If I don't have a free physical register the instruction remains in the issue stage.

Register renaming vs. ROB

More complex to deallocate the register as we have the freelist. The other problem is that the execution of the code is not respected in the name of the registers name in the table(difficulty in knowing the correspondence between registers). Number of register added is random(up to 80).

Multiple issues

Necessary to have the issue logic to handle two or more instructions at once, including possible dependences between instructions
Most fundamental bottleneck in dynamically scheduled superscalar processors

- Need logic to handle issuing every possible combination of dependent instructions in the same clock cycle
- Since the number of possibilities increases with the square of the number of instructions that can be issued in one clock cycle, difficult to implement issue logic for more than 4 instructions
  There is a basic strategy:

1. Assign a reservation station and a reorder buffer entry for every instruction in the next issue bundle
   - if not available only a subset of instructions is considered in sequential order
2. Analyze all dependences among the instructions
3. If an instruction in the bundle depends on an earlier instruction of the same bundle, use the assigned reorder buffer number to update the reservation table for the dependent instruction. All this is done in parallel in a single clock cycle!
4. In addition we need to be able to commit multiple instructions in a single clock cycle