

# Dynamic Scheduling

## Why go out of style?

- expensive hardware for the time (actually, still is, relatively)
- register files grew so less register pressure
- early RISCs had lower CPIs

# Dynamic Scheduling

## Why come back?

- higher chip densities
- greater need to hide latencies as:
  - discrepancy between CPU & memory speeds increases
  - branch misprediction penalty increases from superpipelining
- dynamic scheduling was generalized to cover more than floating point operations
  - handles branches & hides branch latencies
  - hides cache misses
  - can be implemented with a more general register renaming mechanism
- commits instructions in-order to preserve precise interrupts
- processors now issue multiple instructions at the same time
  - more need to exploit ILP

**2 styles:** large physical register file & reorder buffer  
(R10000-style) (PentiumPro-style)

# Precise Interrupts

**Precise interrupts** preserve the model that instructions execute in program-generated order, one at a time

- If a recoverable interrupt occurs, the processor can recover from it

What happens on a precise interrupt:

- ***identify the instruction that caused the interrupt***
- ***let the instructions before faulting instruction finish***
- disable writes for faulting & subsequent instructions
- force trap instruction into pipeline
- trap routine
  - save the state of the executing program
  - correct the cause of the interrupt
  - restore program state
- ***restart faulting & subsequent instructions***

# Register Renaming with A Physical Register File

Register renaming provides a **mapping** between 2 register sets

- **architectural registers** defined by the ISA
- **physical registers** implemented in the CPU
  - hold results of the instructions committed so far
  - hold results of subsequent instructions that have not yet committed
  - more of them than architectural registers
    - $\sim$  issue width \* # pipeline stages between register renaming & commit

# Register Renaming with A Physical Register File

How does it work?:

- an architectural register is mapped to a physical register during a register renaming stage in the pipeline
  - destination registers create mappings
  - source registers use mappings
- operands thereafter are called by their physical register number
  - hazards determined by comparing physical register numbers, not architectural register numbers

# A Register Renaming Example

Code Segment	Register Mapping	Comments
<code>ld r7, 0(r6)</code>	<code>r7 -&gt; p1</code>	<code>p1</code> is allocated
...		
<code>add r8, r9, r7</code>	<code>r8 -&gt; p2</code>	use <code>p1</code> , not <code>r7</code>
...		
<code>sub r7, r2, r3</code>	<code>r7 -&gt; p3</code>	<code>p3</code> is allocated <code>p1</code> is deallocated when <code>sub</code> commits

# Register Renaming with A Physical Register File

Effects:

- eliminates WAW and WAR hazards (*false name* dependences)
- increases ILP

# An Implementation (R10000)

Modular design with regular hardware data structures

Structures for register renaming

- 64 **physical registers** (each, for integer & FP)
- **map tables** for the **current** architectural-to-physical register mapping (separate, for integer & FP)
  - accessed with an architectural register number
  - produces a physical register number
- source operands refer to the latest defined destination register, i.e., the current mappings
- a destination register is assigned a new physical register number from a **free register list** (separate, for integer & FP)



## An Implementation (R10000)

### **Instruction “queues”** (integer, FP & data transfer)

- contains decoded & mapped instructions with the current physical register mappings
  - instructions entered into free locations in the IQ
  - sit there until they are dispatched to functional units
  - somewhat analogous to Tomasulo reservation stations without value fields or valid bits
- used to determine when operands are available
  - compare each source operand of instructions in the IQ to destination values just computed
- determines when an appropriate functional unit is available
- dispatches instructions to functional units

# An Implementation (R10000)

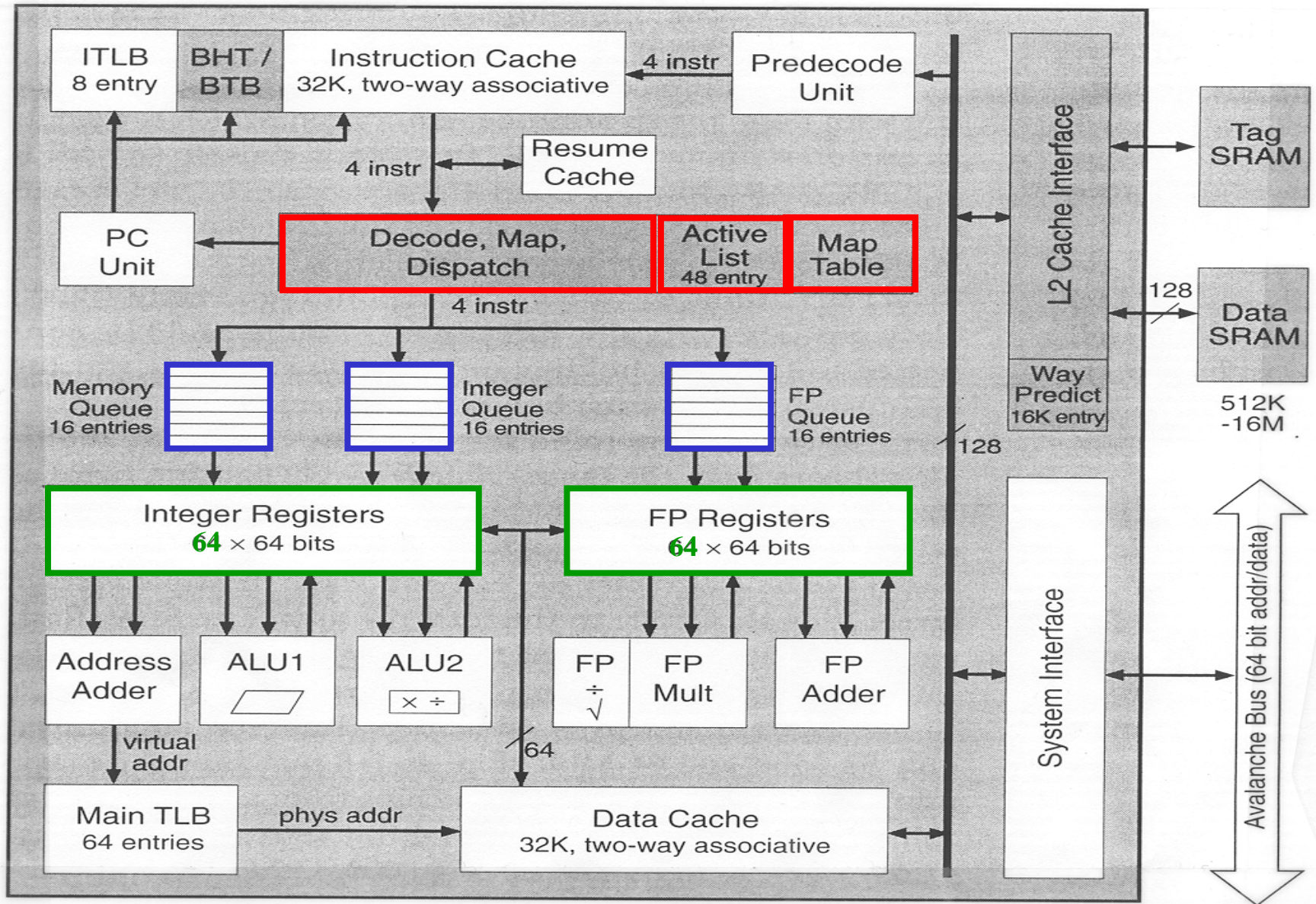
**active list** for all uncommitted instructions

- the mechanism for maintaining precise interrupts
  - instructions entered in program-generated order
  - allows instructions to complete in program-generated order
- instructions removed from the active list when:
  - an instruction commits:
    - the instruction has completed execution
    - all instructions ahead of it have also completed
  - branch is mispredicted
  - an exception occurs
- contains the **previous** architectural-to-physical destination register mapping
  - used to recreate the map table for instruction restart after an exception
- instructions in the other hardware structures & the functional units are identified by their active list location

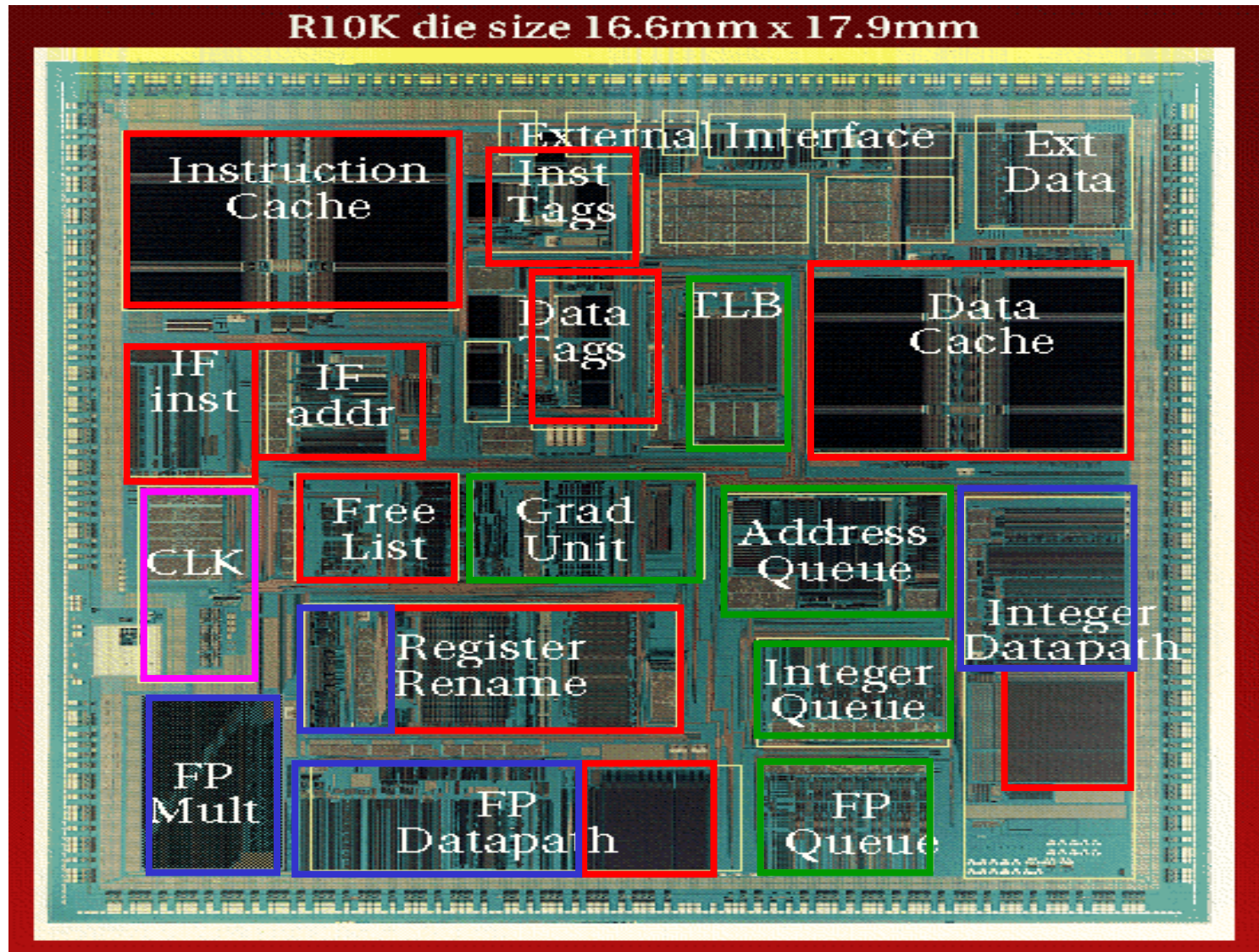
# An Implementation (R10000)

**busy-register table** (integer & FP):

- indicates whether a physical register contains a value
- somewhat analogous to Tomasulo's register status
- used to determine operand availability
  - bit is set when a register is mapped & leaves the free list (not available yet)
  - cleared when a FU writes the register (now there's a value)

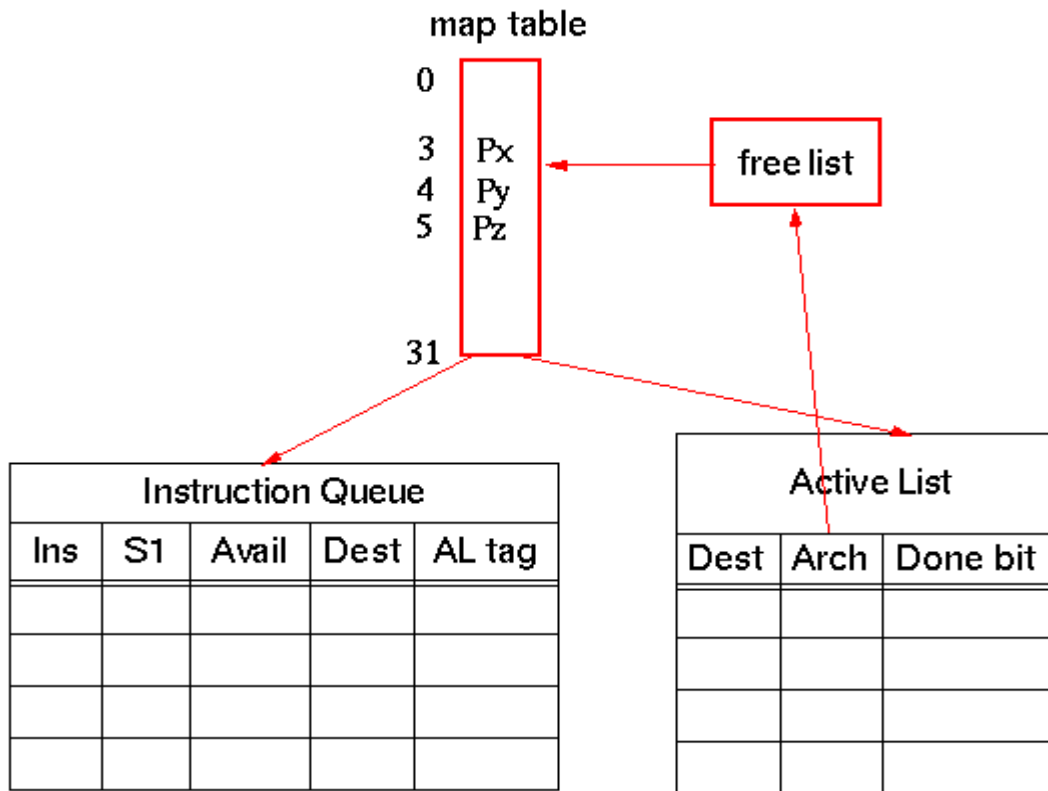


# R10000 Die Photo



# The R10000 in Action 1

→ ld A3, #(reg) arch register **A3 defined**  
 potential multi-cycle  
 add A4, A3, reg arch register **A3 used**  
 sub A3, reg, reg arch register **A3 redefined**  
 name dependence  
 or A5, A3, reg arch register **A3 used**



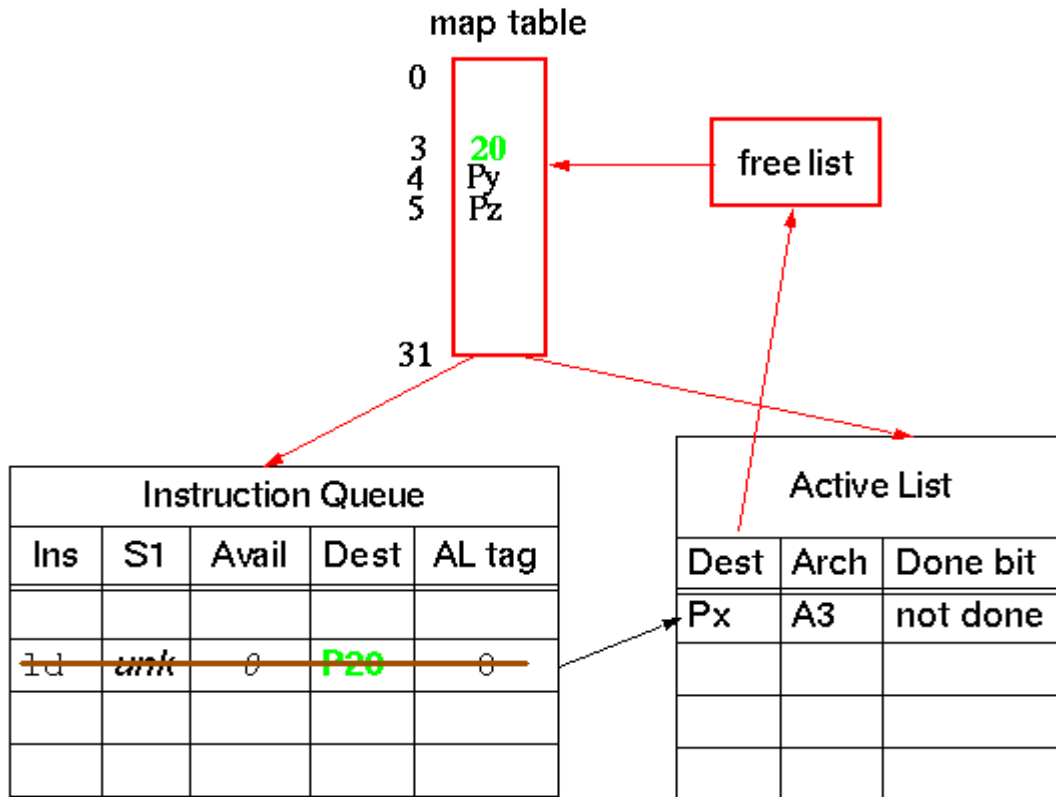
# The R10000 in Action 2

→ ld A3, #(reg) arch register **A3 defined**  
potential multi-cycle

add A4, A3, reg

sub A3, reg, reg

or A5, A3, reg



# The R10000 in Action 3

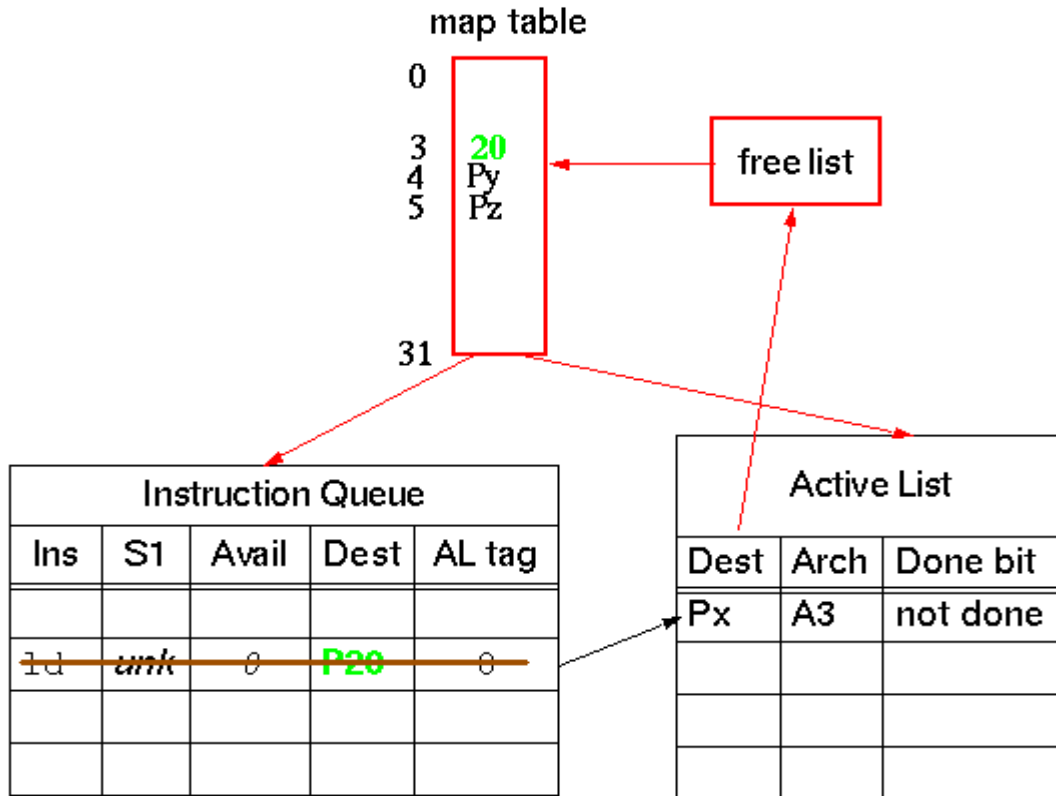
ld A3, #(reg) arch register **A3 defined**  
potential multi-cycle



add A4, A3, reg

sub A3, reg, reg

or A5, A3, reg

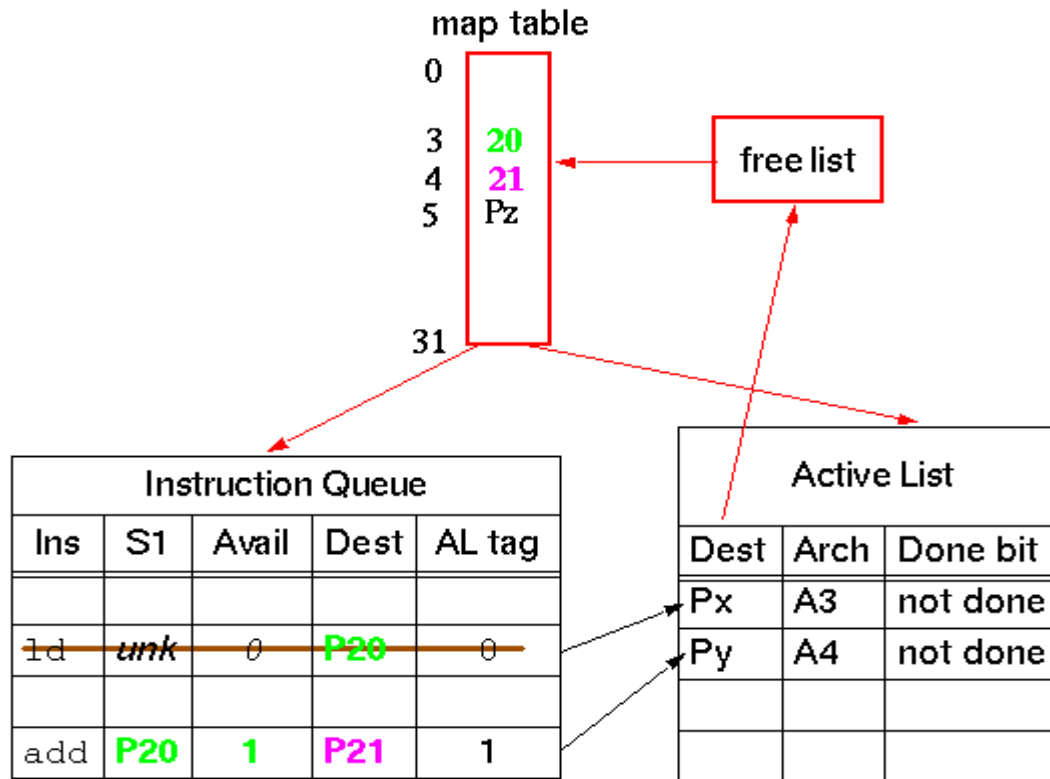




# The R10000 in Action 4

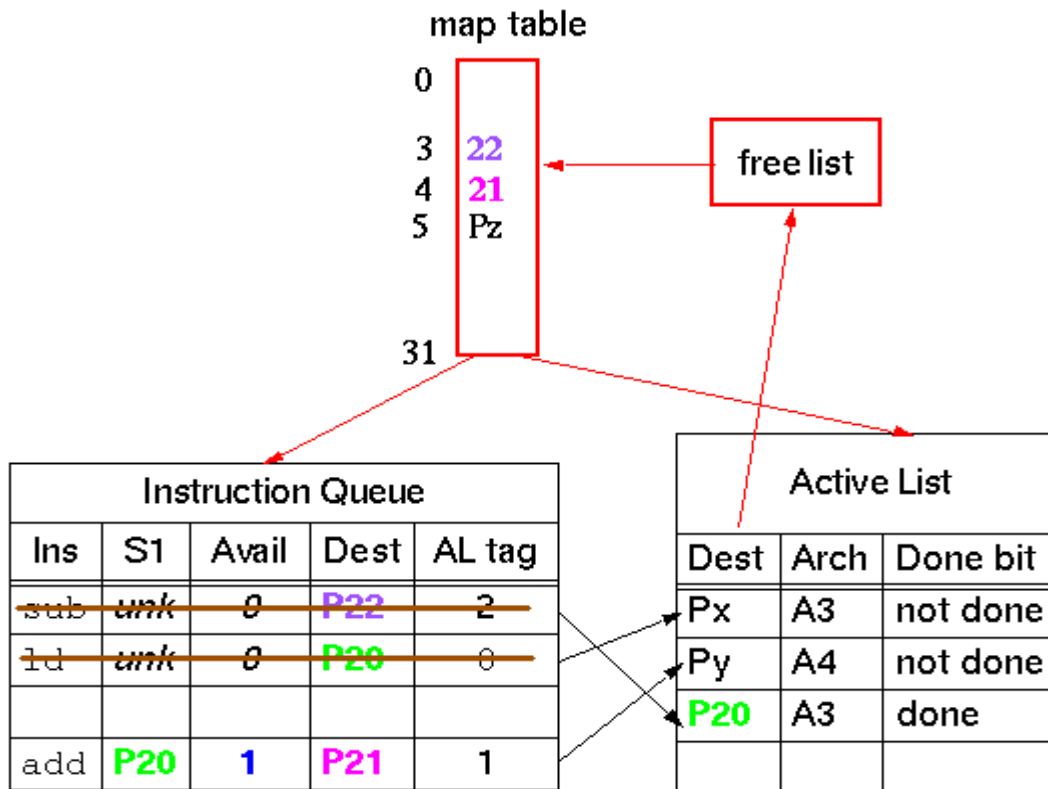
```

ld    A3, #(reg)    arch register A3 defined
                        potential multi-cycle
add   A4, A3, reg    arch register A3 used
→ sub  A3, reg, reg
or    A5, A3, reg
  
```



# The R10000 in Action 5

```
ld    A3, #(reg)    arch register A3 defined
                        potential multi-cycle
add   A4, A3, reg    arch register A3 used
sub   A3, reg, reg    arch register A3 redefined
                        name dependence
→ or  A5, A3, reg
```



# The R10000 in Action 5 : Interrupts 1

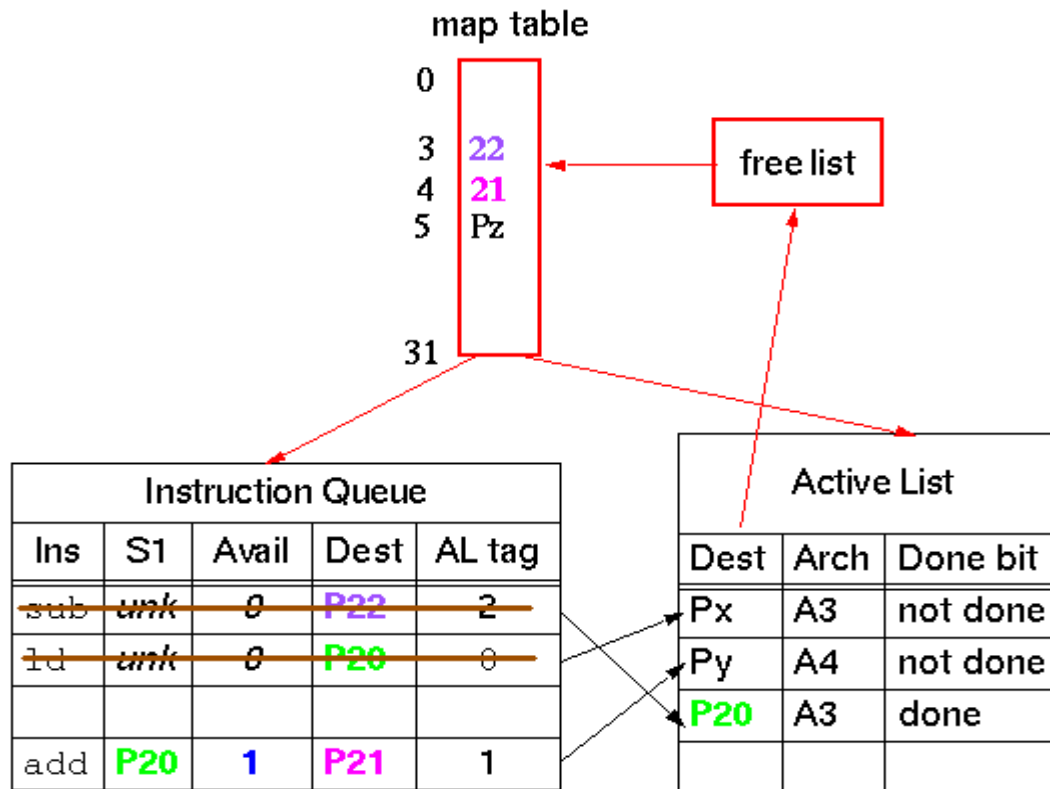
```

ld      A3, #(reg)      arch register A3 defined
                                potential multi-cycle

add     A4, A3, reg     arch register A3 used

sub     A3, reg, reg    arch register A3 redefined
                                name dependence

→ or    A5, A3, reg
  
```



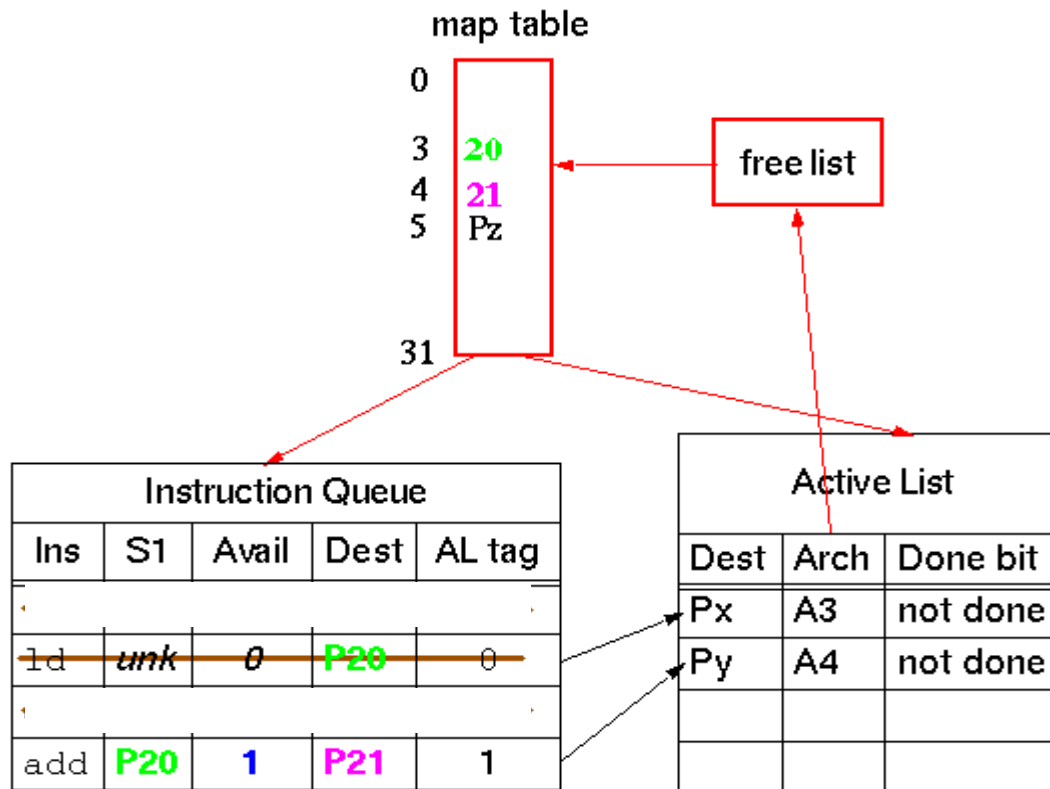
# The R10000 in Action: Interrupts 2

ld A3, #(reg) arch register **A3 defined**  
 potential multi-cycle

add A4, A3, reg arch register **A3 used**

→ sub A3, reg, reg arch register **A3 redefined**  
 name dependence

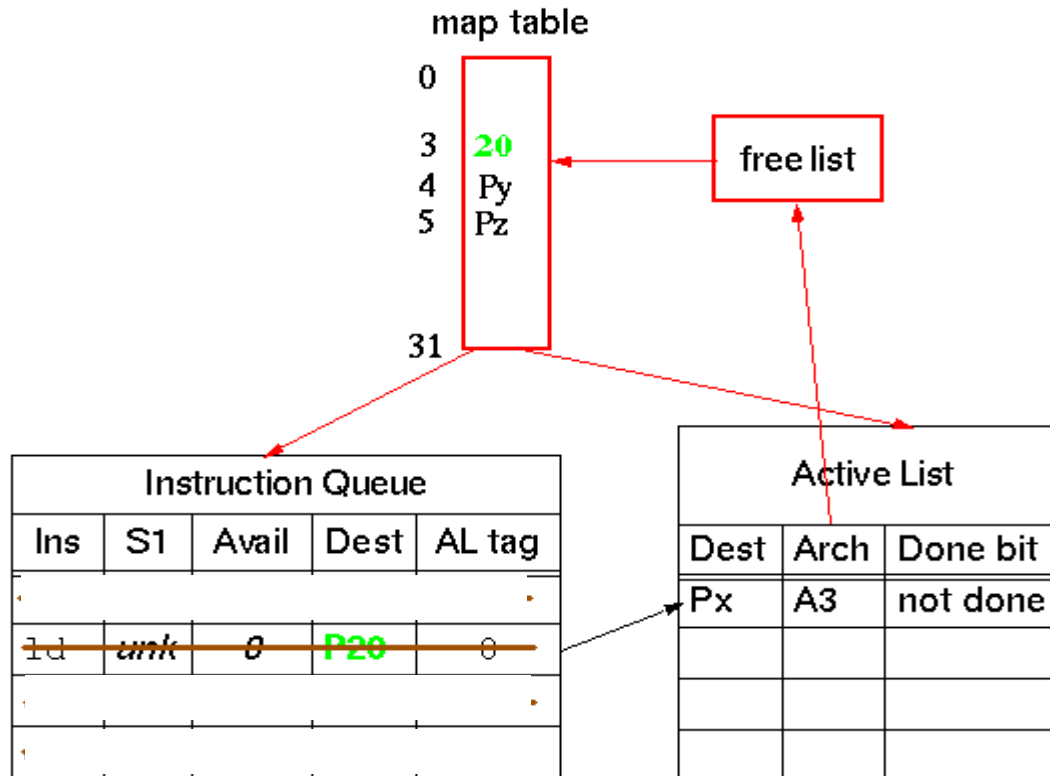
or A5, A3, reg arch register **A3 used**



# The R10000 in Action: Interrupts 3

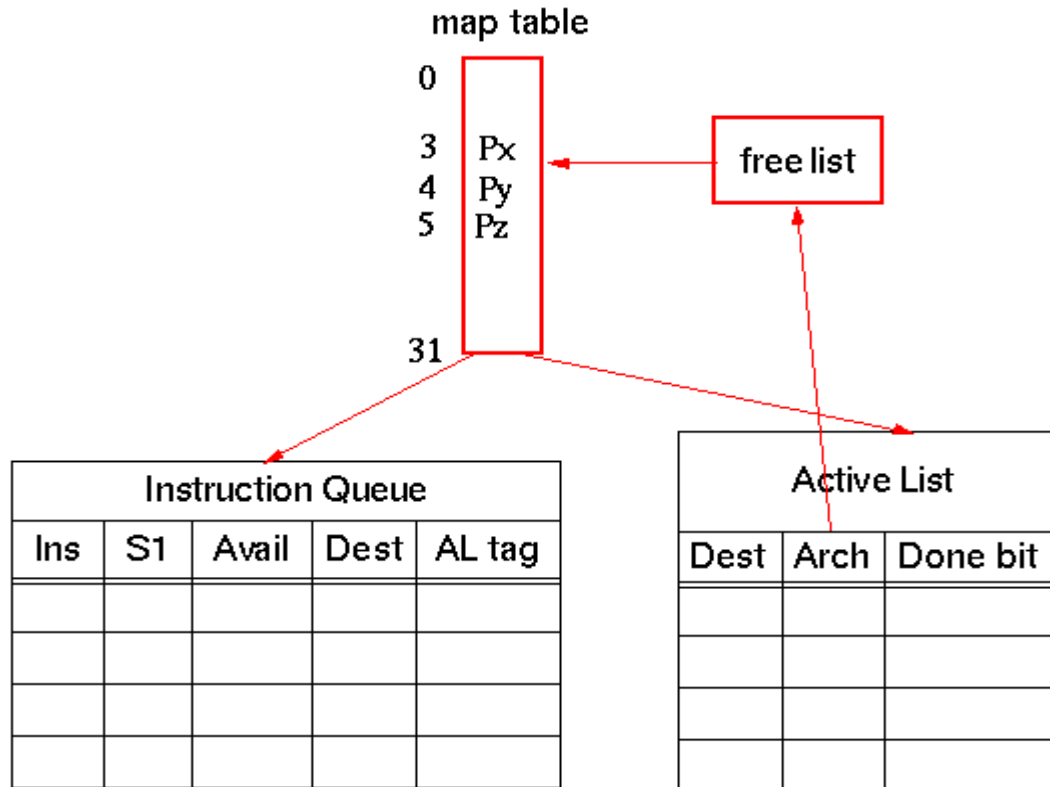
```

ld    A3, #(reg)    arch register A3 defined
                        potential multi-cycle
→ add  A4, A3, reg   arch register A3 used
sub   A3, reg, reg   arch register A3 redefined
                        name dependence
or    A5, A3, reg    arch register A3 used
  
```



# The R10000 in Action: Interrupts 4

→ ld A3, #(reg) arch register **A3 defined**  
 potential multi-cycle  
 add A4, A3, reg arch register **A3 used**  
 sub A3, reg, reg arch register **A3 redefined**  
 name dependence  
 or A5, A3, reg arch register **A3 used**



# R10000 Execution

## **In-order issue** (have already fetched instructions)

- rename architectural registers to physical registers via a map table
- detect structural hazards for instruction queues (integer, memory & FP) & active list
- issue up to 4 instructions to the instruction queues

## **Out-of-order execution** (to increase ILP)

- reservation-station-like instruction queues that indicate when an operand has been calculated
  - each instruction monitors the setting of the busy-register table
- set busy-register table entry for the destination register
- detect functional unit structural & RAW hazards
- dispatch instructions to functional units

## **In-order commit** (to preserve precise interrupts)

- this & previous program-generated instructions have completed
- physical register in previous mapping returned to free list
- rollback on interrupts