

# The Pentium® II/III Processor “Compiler on a Chip”

Ronny Ronen  
Senior Principal Engineer  
Director of Architecture Research  
Intel Labs - Haifa

Intel Corporation

Tel Aviv University  
January 18, 2005

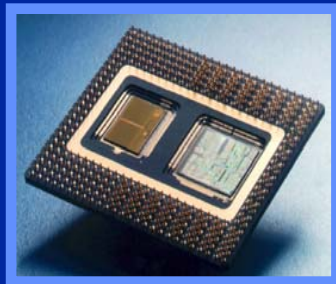
# Agenda

- Goal, Expectations...
- General Information
- $\mu$ architecture basics
- Pentium<sup>®</sup> Pro Processor  $\mu$ architecture
- SW aspects

# Technology Profile

## Pentium Pro - 1995

- Core @200MHz
- 256K L2 on package, @200MHz
- Performance:
  - 8.09 SPECint95
  - 6.70 SPECfp95
- 0.35  $\mu\text{m}$  BiCMOS
- 5.5M transistors
- 195 sq mm (14x14)
- 3.3V, 11.2A
- 28.1W / 35.0W



## Pentium-II - 1998

- Core @333MHz
- 512KB L2 in SEC @167MHz
- Performance:
  - 12.8 SPECint95
  - 9.14 SPECfp95 (P55C: 7.12/5.21)
- 0.25  $\mu\text{m}$  CMOS process
- 7.5M transistors



## Pentium-III - 1999

- Core @600MHz
- 512KB L2 @ ???MHz
- Performance:
  - 24.0 SPECint95
  - 15.9 SPECfp95
- 0.25  $\mu\text{m}$  CMOS process
- ???M transistors



# Technology Profile (cont.)

- Pentium-III – 2000 (Coppermine)
- Core @1000MHz
- 256KB L2 on chip @ 1000MHz
- Performance:
  - ▣ >46 SPECint95
  - ▣ >20 SPECfp95
- 0.18  $\mu\text{m}$  CMOS process
- ~20M transistors

- Pentium M Processor 2003 (Banias)
- Core @1800MHz
- 1024KB L2 on chip @ 1800MHz
- Performance (estimated):
  - ▣ >80 SPECint95
  - ▣ >50 SPECfp95
- 0.13  $\mu\text{m}$  CMOS process

intel® ~77M transistors

- Pentium-III - 2002 (Tualatin)
- Core @1400MHz
- 512KB L2 on chip @ 1400MHz
- Performance (estimated):
  - ▣ >60 SPECint95
  - ▣ >30 SPECfp95
- 0.13  $\mu\text{m}$  CMOS process
- ~44M transistors

**Rough**

- Pentium M Processor 2004 (Dothan)
- Core @2000MHz
- 2048KB L2 on chip @ 2000MHz
- Performance (estimated):
  - ▣ >90 SPECint95 (est.)
  - ▣ >60 SPECfp95 (est.)
- 0.09  $\mu\text{m}$  CMOS process
- ~127M transistors

**Rough**

# Terminology

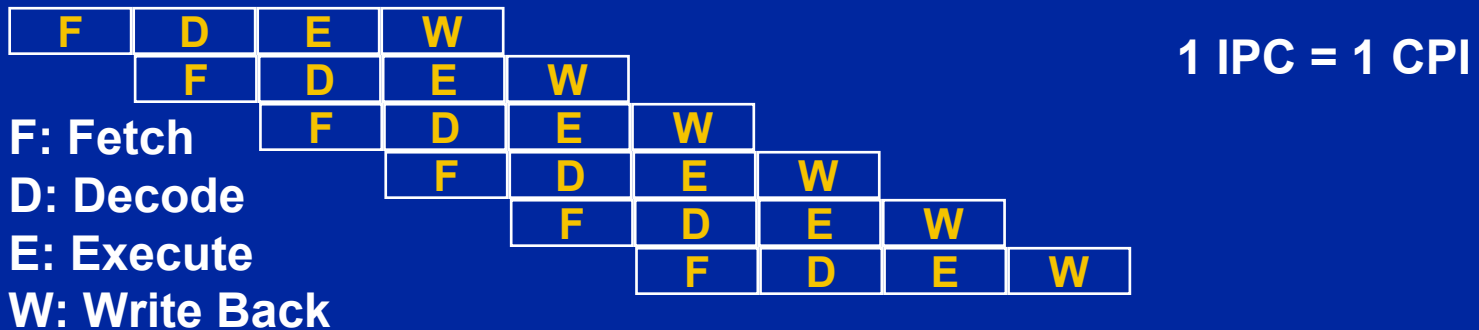
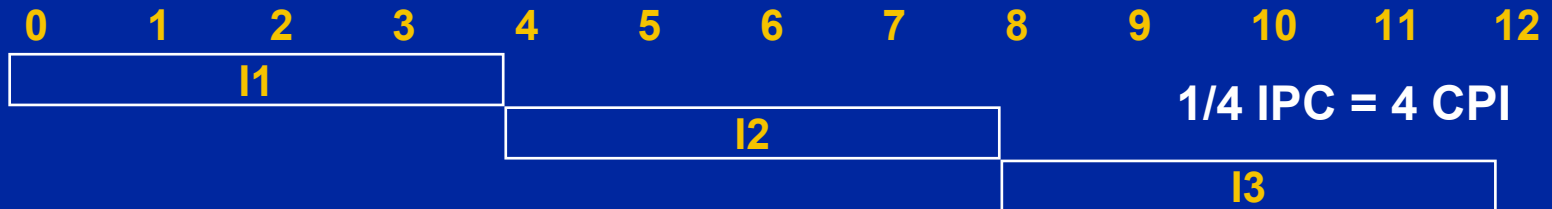
- Intel Architecture
- Pipeline, Super Scalar
- Branch Prediction
- Speculative Execution
- Dynamic Scheduling
- Data dependency
- Register Renaming
- Out Of Order
- Re-order Buffer & Memory Order Buffer
- Reservation Stations
- Micro-Operations

# Intel Architecture (X86)

- **8 registers only**
  - Can be partially accessed
  - ⇒ Many memory accesses, short life time
- **One set of condition codes**
  - Modified by most ALU operations
  - Various operations affect various flags
  - ⇒ Short Generate/Use distance
- **Explicit stack**
  - Push/Pop/Call/Return operations
- **Variable length instructions**
  - Implicit operands

# Pipeline

- Break the work to smaller pieces

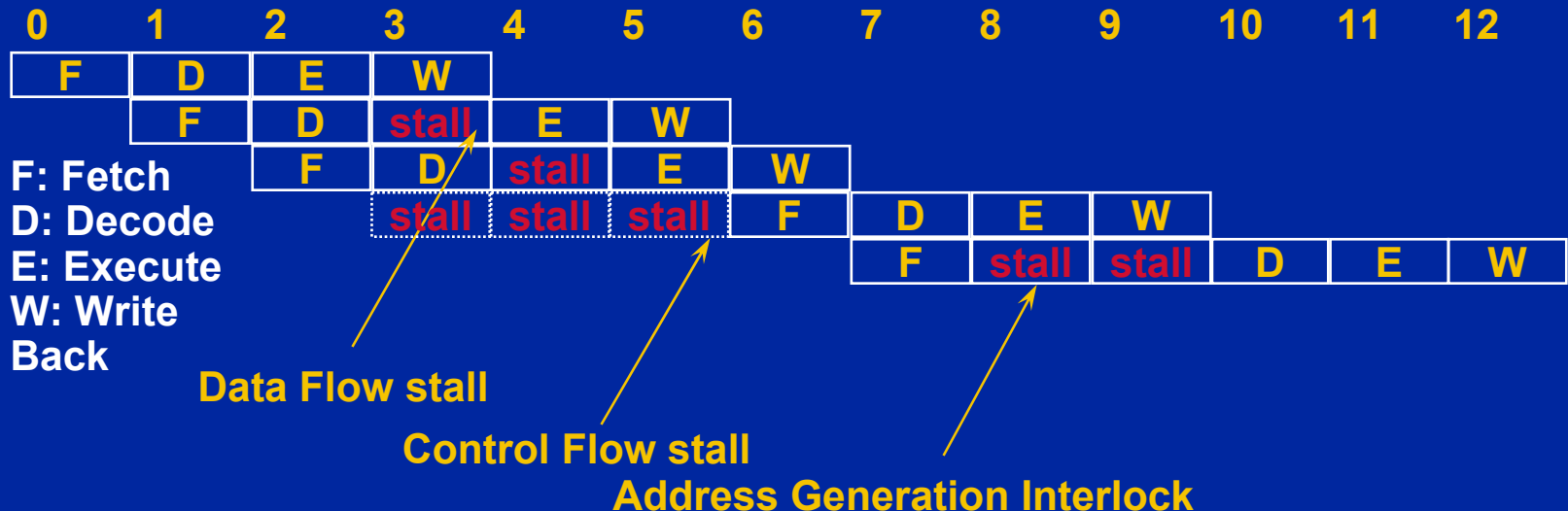


- Increased throughput

- increased # of completed instructions per cycle
- Number of stages varies
  - Small: 4-5 (Pentium), “Superpipeline” ~14 (Pentium Pro)

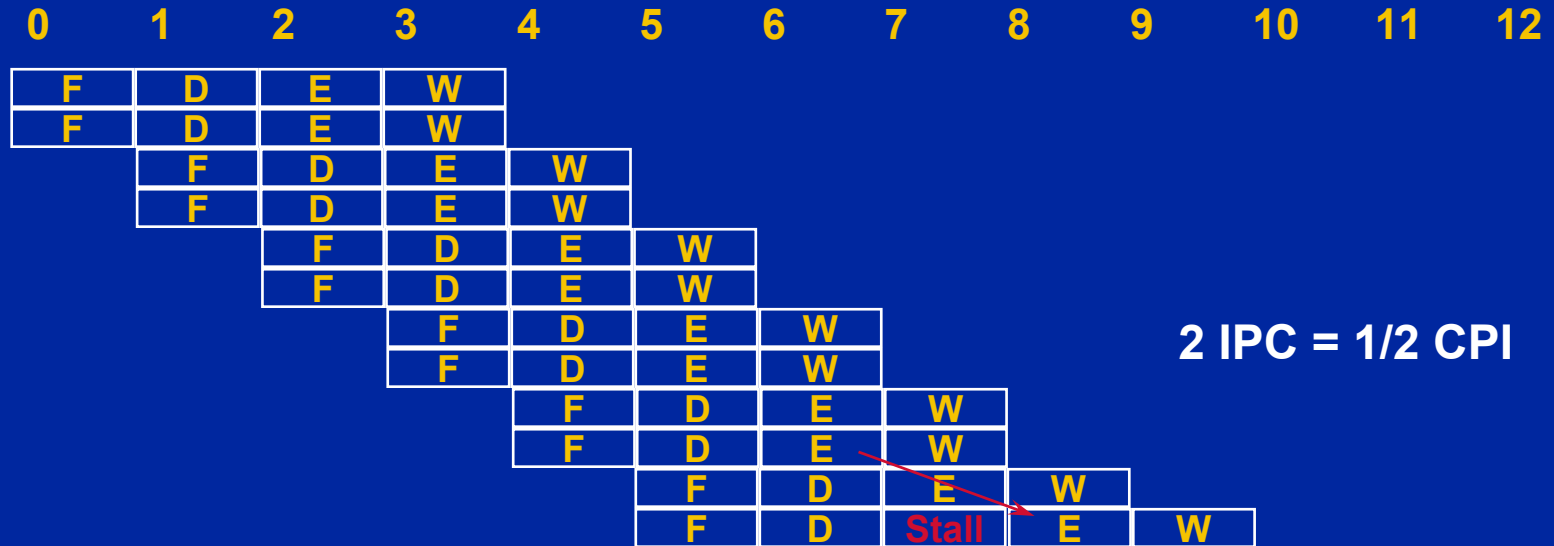
# Pipeline Stalls

- But there are “stalls” in the pipeline
  - Data flow dependency (instructions output/input)
    - Solved by: bypasses, renaming
  - Control flow dependencies
    - Solved by branch prediction
  - Other (Cache misses, long latency instructions)



# SuperScalar

- Performs more in a single cycle



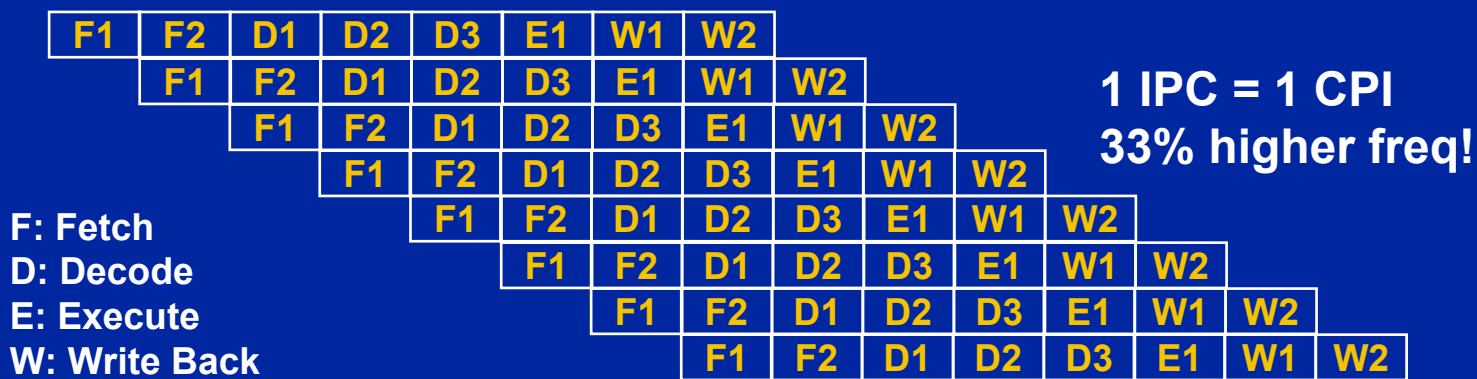
- Ideally, can multiply the throughput
  - but stall penalty is increased

# Super Pipeline

- Split to shorter stages - allows higher frequency

Old clk = 0 1 2 3 4 5 6 7 8 9 10 11 12

New clk = 0 1 2 3 4 5 6 7 8 9 10 11 12



- Ideally, can (again) multiply the throughput, but
  - Stall penalties do not scale (e.g., control flow stall, cache misses)
  - Clock setup/hold reduces the amount of net cycle time more - each instruction takes longer!
- ⇒ In the example above: 2X stages, but performance gain is <<33%

# Out Of Order Execution

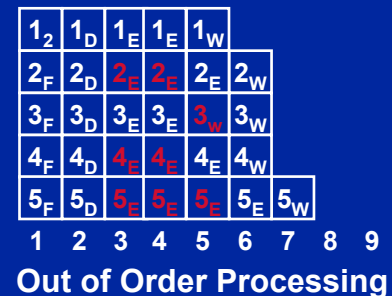
- So far - instructions were processed in their program order.
  - Parallelism is limited.
- OOO: Instructions are executed based on “*data flow*” rather than program order

Before: src -> dest

- (1) load (r10), r21
- (2) mov r21, r31 (2 depends on 1)
- (3) load a, r11
- (4) mov r11, r22 (4 depends on 3)
- (5) mov r22, r23 (5 depends on 4)

After:

- (1) load (r10), r21; (3) load a, r11;  
    <wait for loads to complete>
- (2) mov r21, r31; (4) mov r11, r22;  
                  (5) mov r22, r23;



In Order vs OOO execution.

Assuming:

- Unlimited resources
- 2 cycles load latency

• Usually highly superscalar



# Cache - Motivation & Principle



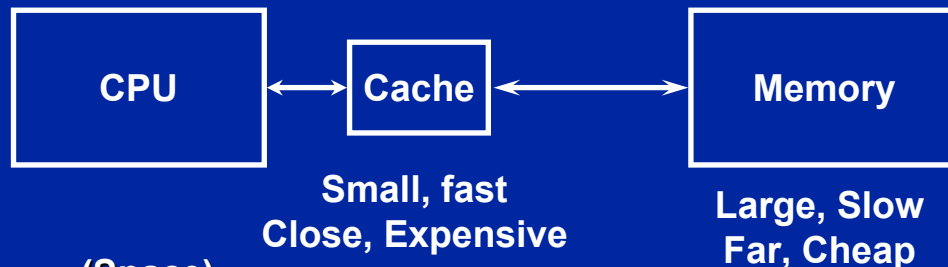
- Memory consumption is growing about 2X every 2 years
  - Typical size: (Y2000) 64M-128M, (Y2002) 128M-256M
- CPU speed grows faster than memory and buses
  - CPU/Bus grew from 1:1 to 6:1, and still growing

486	Pentium	P-II	P-III	P4
25-66MHz	66-233MHz	200-450MH	0.5-1.33GHz	1.4-2.4GHz
33MHz	66MHZ	66-100MHz	133-200MHz	400MHz

- Memory: DRAM: 60-100ns (“10-16MHz”), Cost: <10\$ per 1M  
SRAM is faster but much more expensive  
Bandwidth: SDRAM 100-133MHz\*8B; DDR 200-400MHz\*8B; RDR 800MHz\*2B
- ⇒ *Memory becomes the bottleneck for both instructions and data!*
- Slow or expensive

- Solution: Cache - A Small, Fast, Close memory
  - Serves as a buffer between CPU and main memory

- Contains copy of a portion of the main memory
  - Small in size
  - Dynamically changed

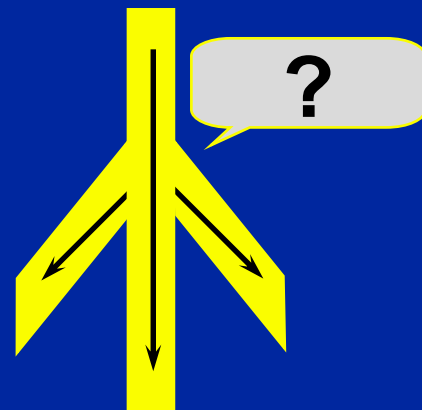


- Exploit space and time locality:

- Code is fetched sequentially (Space)
- Code is re-executed (loops, procedures) (Time)
- Access close or previous data (Space, Time)

# Branch Prediction

- Goal - ensure enough instruction supply by correct prefetching
- up to 486 - prefetcher assumed *fall-through*
  - Lose on unconditional branch (e.g., call)
  - Lose on frequently taken branches (e.g., loops)
- Branch prediction
  - Predict branch *taken/not taken*
  - Predict the branch target address



# Branch Prediction (cont.)

- **Implementation**

- Use history (private or global) to predict direction (simple Lee&Smith, advanced Yeh&Patt)
- Target address taken from table (faster) or from the instruction (slower)
- Table updated first based on prediction, later based on actual execution.

- **Misprediction cost varies (high on PPro)**

- **Current prediction rate: ~92% - 95%**

**~60-100 instructions between mispredictions\***

\* Assuming branch every 5 instructions

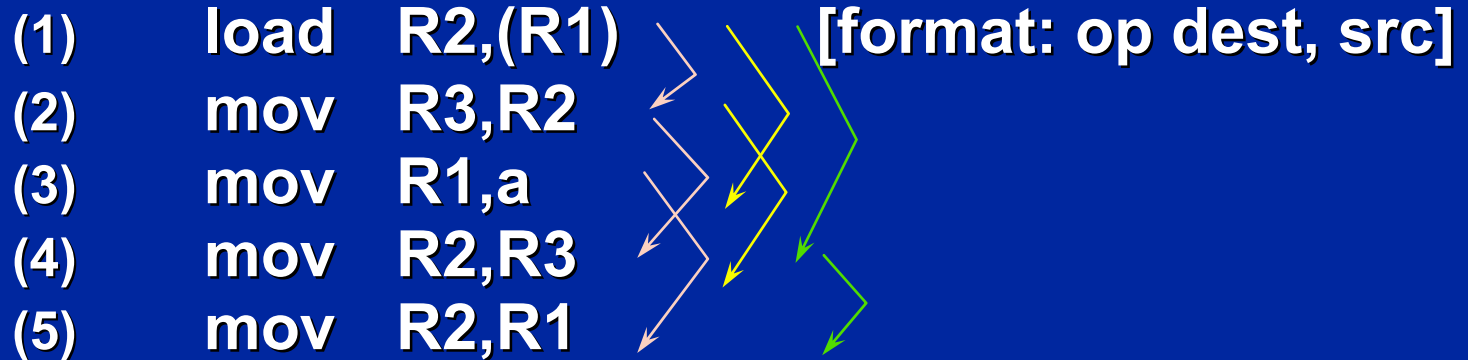
# Speculative Execution

- Execution of instructions from a predicted (yet unsure) path.  
Eventually, path may turn wrong.
- Advantages:
  - Ensure instruction supply
  - Allow scheduling window
- Issues:
  - Misprediction cost
  - Misprediction recovery

# Dynamic Scheduling

- Scheduling instructions at run time, by the HW, and not at compile time, by the SW
- Advantages:
  - Works on the dynamic instruction flow:  
Can schedule across procedures, modules...
  - Can see dynamic values (memory addresses)
  - Can accommodate varying latencies
- Disadvantages
  - Can schedule within a limited window only
  - Should be fast - cannot be too smart

# Data Dependency



- True dependency

(R2:1>2, R3:2>4, R1:3>5)

- False dependencies

- *Anti dependency*

(R1:1>3, R2: 2>4)

- *Output dependency*

(R2:1>4,R2:4>5)

# Register Renaming

	before		after	mapping
(1)	load R2,(R1)	load	r21,(r10)	[R2->r21]
(2)	mov R3,R2	mov	r31,r21	[R3->r31]
(3)	mov R1,a	mov	r11,a	[R1->r11]
(4)	mov R2,R3	mov	r22,r31	[R2->r22]
(5)	<b>add</b> R2,R1	<b>add</b>	r23,r11,r22	[R2->r23]

- Remove false dependencies
- Remove architecture limit for # of regs
- Help speculative execution
  - Renamed register are kept until speculation is verified to be correct

# Out Of Order Execution

- Execute instructions based on “*data flow*” rather than program order

Before:

- (1) load r21,(r10)
- (2) mov r31,r21                   (2 depends on 1)
- (3) mov r11,a
- (4) mov r22,r31                   (4 depends on 2)
- (5) mov r23,r11                   (5 depends on 3)

After:

- (1) load r21,(r10);   (3) mov   r11,a;
- (2) mov r31,r21;    (5) mov   r23,r11;
- (4) mov r22,r31;

# Out Of Order (cont.)

- Advantages

- Help exploit *Instruction Level Parallelism* (ILP)
- Help cover latencies (e.g., cache miss, divide)
- Superior/complementary to compiler scheduler
  - Dynamic instruction window
  - Can use more than 8 registers

- Complex microarchitecture

- Complex scheduler
- Requires reordering mechanism (*retirement*) in the back-end for:
  - Precise interrupt resolution
  - Misprediction/speculation recovery
  - Memory ordering

# Re-order Buffer (ROB)

- Mechanism for renaming and retirement
- Table contains in-order instructions
  - Instructions are entered in order
  - Registers renamed by the entry#
  - Once assigned: execution order unimportant
  - After execution: entries marked “*executed*”
  - An executed entry can be “*retired*” once all prior instruction have retired. That is:
    - Update “*real registers*” with value of renamed regs
    - Update memory
    - Leave the ROB

# Reservation Station(s)

- Pool(s) of all “not yet executed” instructions
- Maintains operands status “*ready/not-ready*”
- Each cycle, executed instructions make more operands “*ready*”
- Instructions whose all operands are “*ready*” can be “*dispatched*” for execution
- Dispatcher chooses which of the “*ready*” instructions will be executed next.

# Memory Order Buffer (MOB)

- Idea - allow out of order among memory operations
- Problem- Memory dependencies cannot fully resolved statically (memory disambiguation)
  - store r1,a; load r2,b => can advance load before store
  - store r1,[r3]; load r2,b => load should wait till r3 is known
- Structure similar in concept to ROB
- Every access is allocated an entry.  
Address & data (for stores), are updated when known
- Load is checked against all previous stores:
  - Waits if store to same address exist, but data not ready
  - If store data exists, just use it
  - Waits if store to unknown address exists
  - No address collision - go to memory

# Dynamic Execution

- **Combination of three techniques:**
- **Multiple Branch prediction**
- **Out Of Order execution: Dataflow analysis**
- **Speculative Execution**

**How does this machine really work?**

# OOO demo

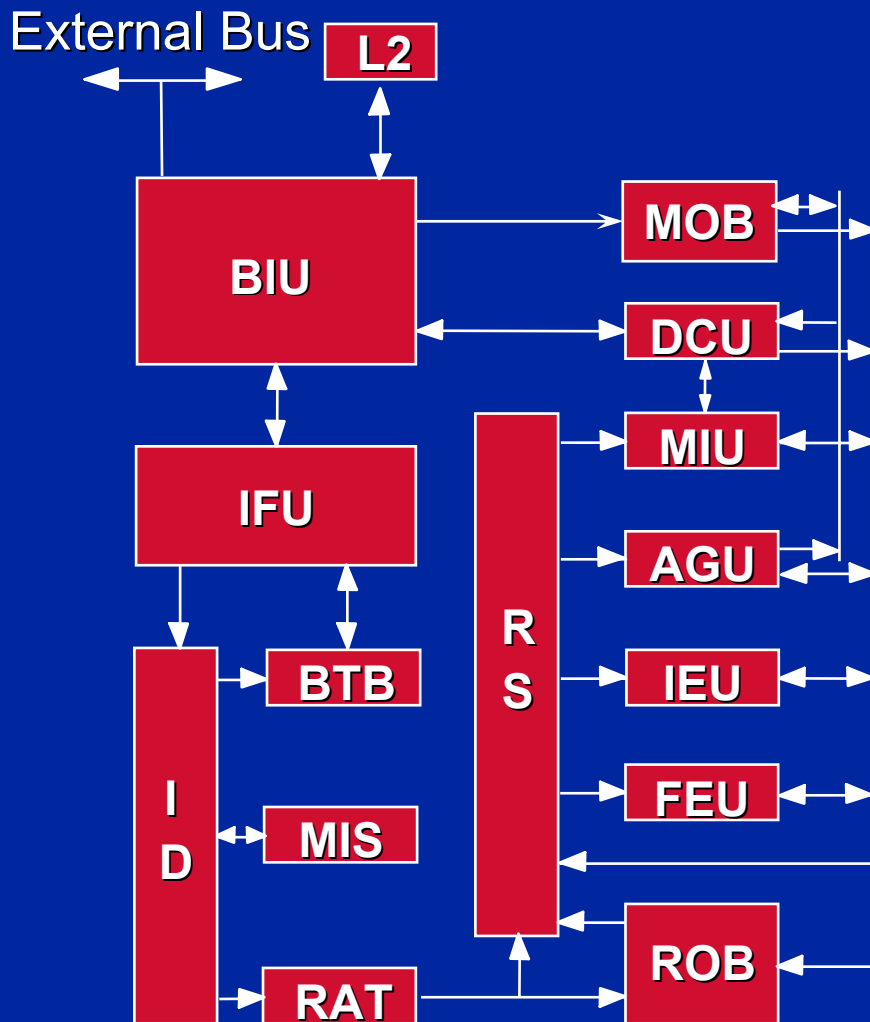
- **The "Pentium® Pro Processor Microarchitecture Overview" tutorial**
  - <http://developer.intel.com/vtune/cbts/cbts.htm>
  - <http://developer.intel.com/vtune/cbts/pproarch/clikngo.htm>

# Micro Operations (Uops)

- Each “CISC” inst is broken into one or more uops
  - Simplicity:
    - Each uop is (relatively) simple
    - Canonical representation of src/dest (3 src, 2 dest)
  - Increased ILP
    - e.g., *pop eax* becomes *esp1<-esp0+4, eax1<-[esp0]*
- Typical uop count (it is not necessarily cycle count!)

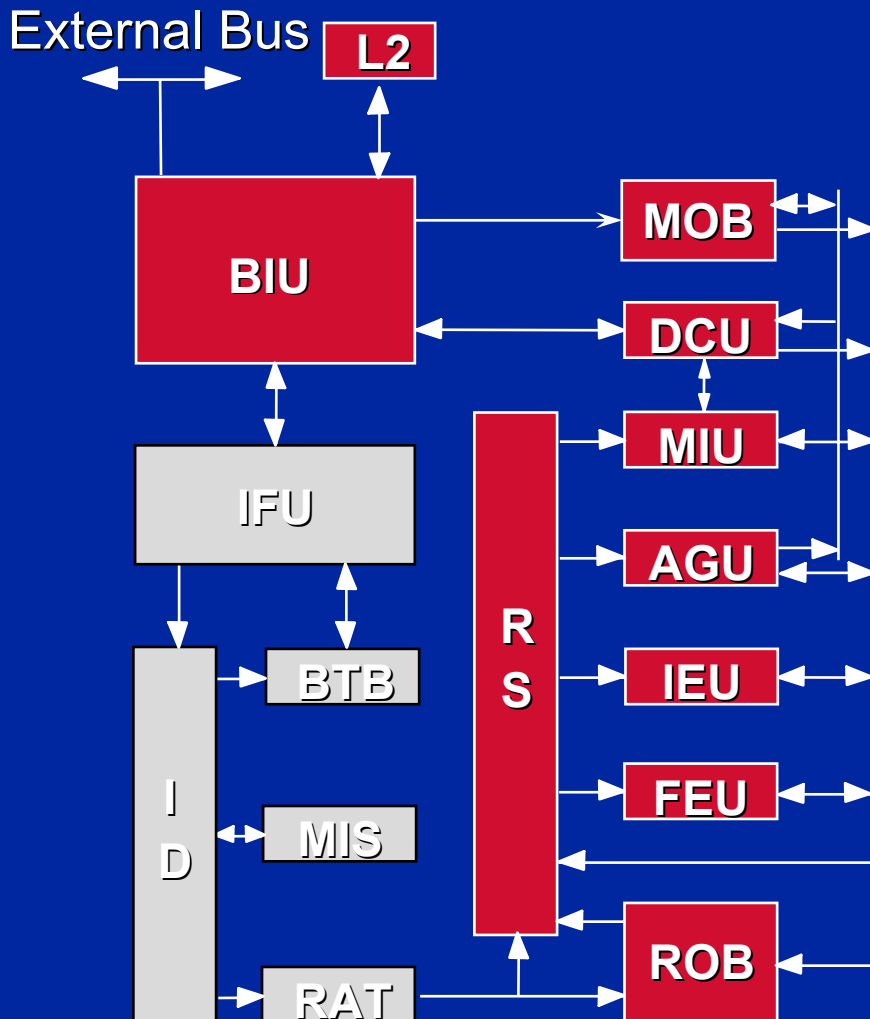
Reg-Reg ALU/Mov inst:	1 uop
Mem-Reg Mov (load)	1 uop
Mem-Reg ALU (load + op)	2 uops
Reg-Mem Mov (store)	2 uops (st addr, st data)
Reg-Mem ALU (ld + op + st)	4 uops
Microcode	Varies
- Mainly an X86 artifact

# CPU Microarchitecture



- In-Order Front End
  - BIU: Bus Interface Unit
  - IFU: Inst. Fetch Unit (includes IC)
  - BTB: Branch Target Buffer
  - ID: Instruction Decoder
  - MIS: Micro-Instruction Sequencer
  - RAT: Register Alias Table
- Out-of-order Core
  - ROB: Reorder Buffer
  - RRF: Real Register File
  - RS: Reservation Stations
  - IEU: Integer Execution Unit
  - FEU: Floating-point Execution Unit
  - AGU: Address Generation Unit
  - MIU: Memory Interface Unit
  - DCU: Data Cache Unit
  - MOB: Memory Order Buffer
  - L2: Level 2 cache
- In-Order Retire

# Microarchitecture



## ● In-Order Front End

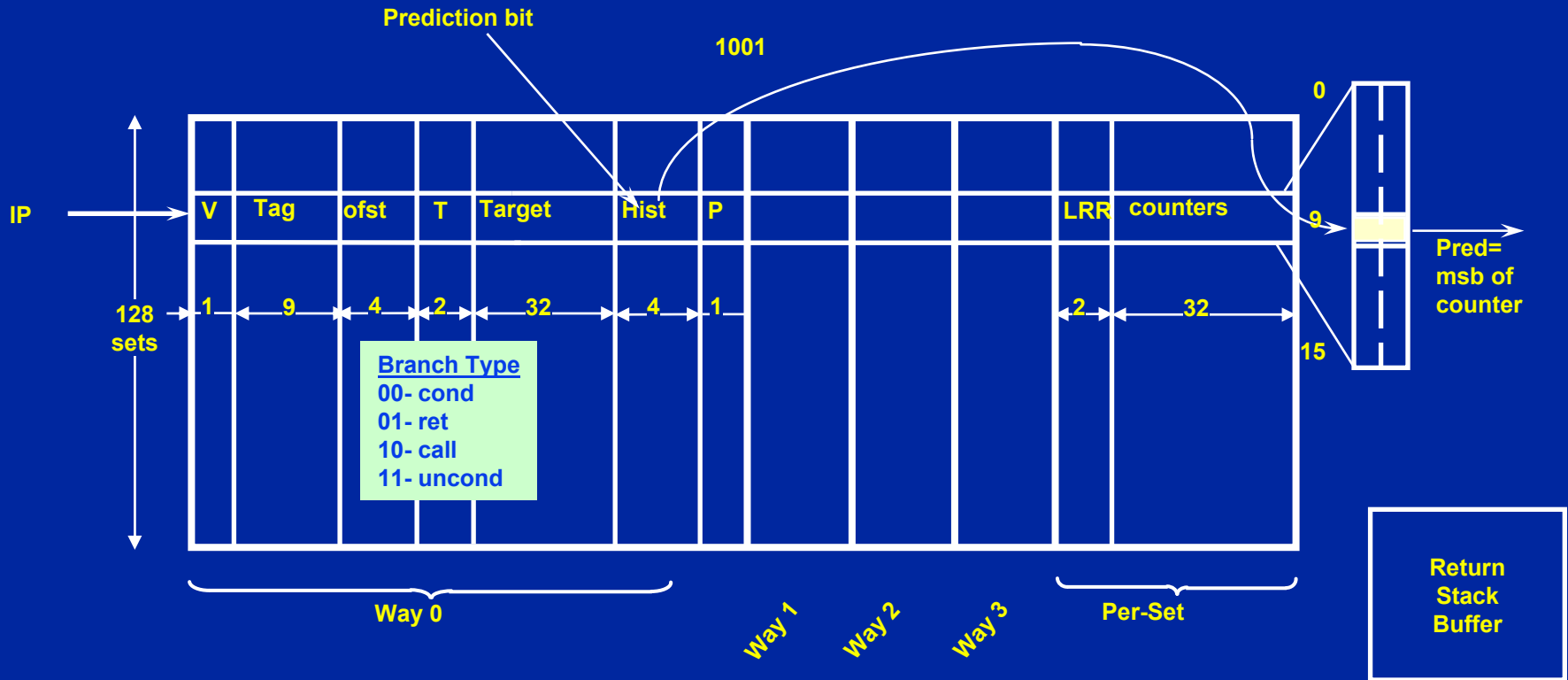
- **BTB**: predicts the address of the next instruction to be fetched
- **IFU**: fetches bytes from the instruction cache (or L2, or memory)
- **ID**: Decodes instructions and converts them to uops (up to 3 uops/cycle).
- **MIS**: Produces uops for complex instructions.
- **RAT**: Register Alias Table

# Branch Prediction

- Implementation
  - Use local history to predict direction
  - Need to predict multiple branches
  - ⇒ Need to predict branches before previous branches are resolved
  - ⇒ Branch history updated first based on prediction, later based on actual execution (speculative history).
  - Target address taken from BTB
- Prediction rate: ~92%
  - ~60 instructions between mispredictions (assuming 1 branch per 5 inst. on average)
  - High prediction rate is very crucial for long pipelines
  - Especially important for OOOE, speculative execution:
    - On misprediction all instructions following the branch in the instruction window are flushed
    - Effective size of the window is determined by prediction accuracy.
- RSB used for Call/Return pairs
- Totally re-done on Banias!

# The P6 BTB

- 2-level, local histories, per-set counters
- 4-way set associative: 512 entries in 128 sets



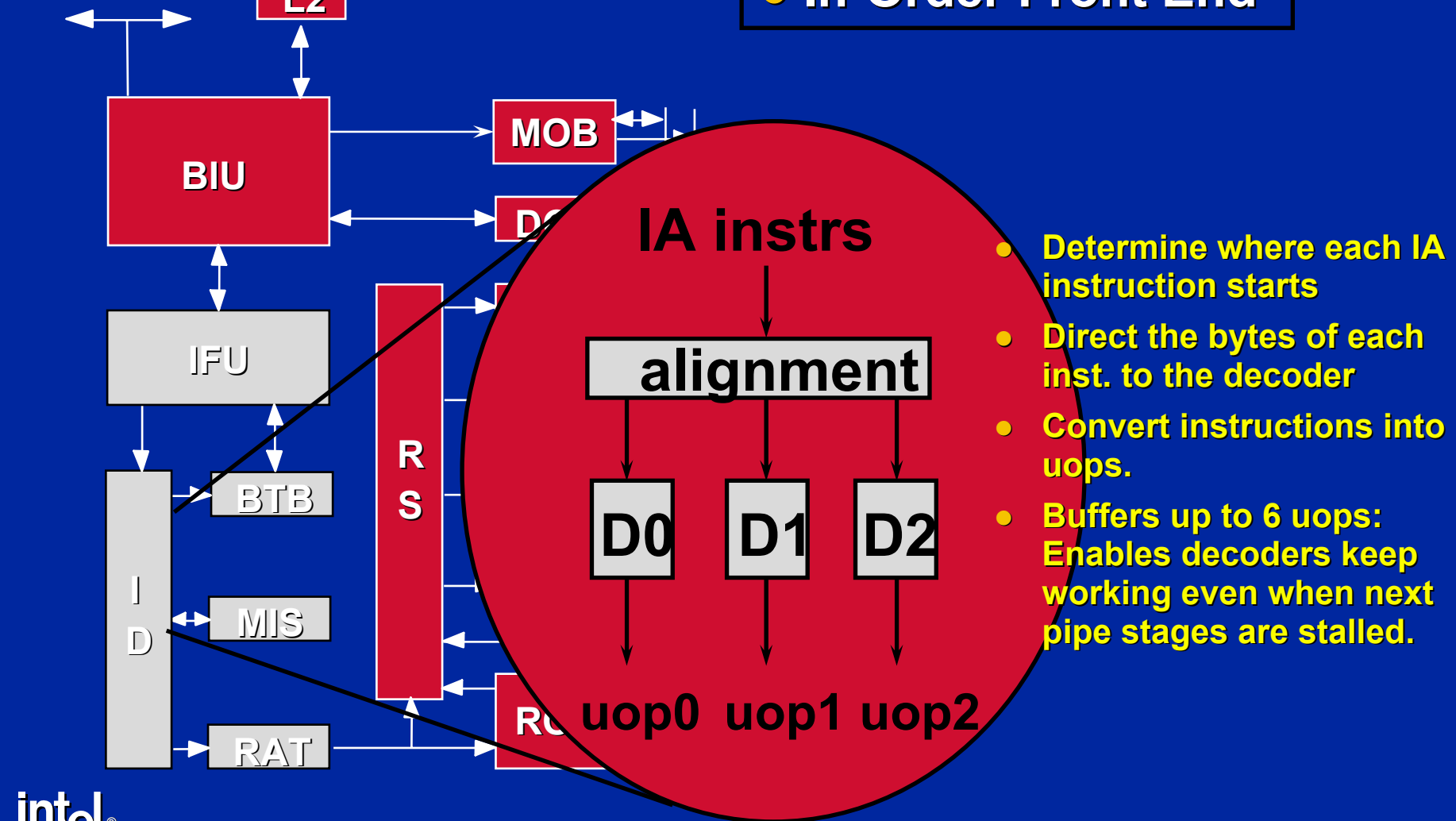
- Up to 4 branches can have a tag match

# Microarchitecture

External Bus

L2

## ● In-Order Front End



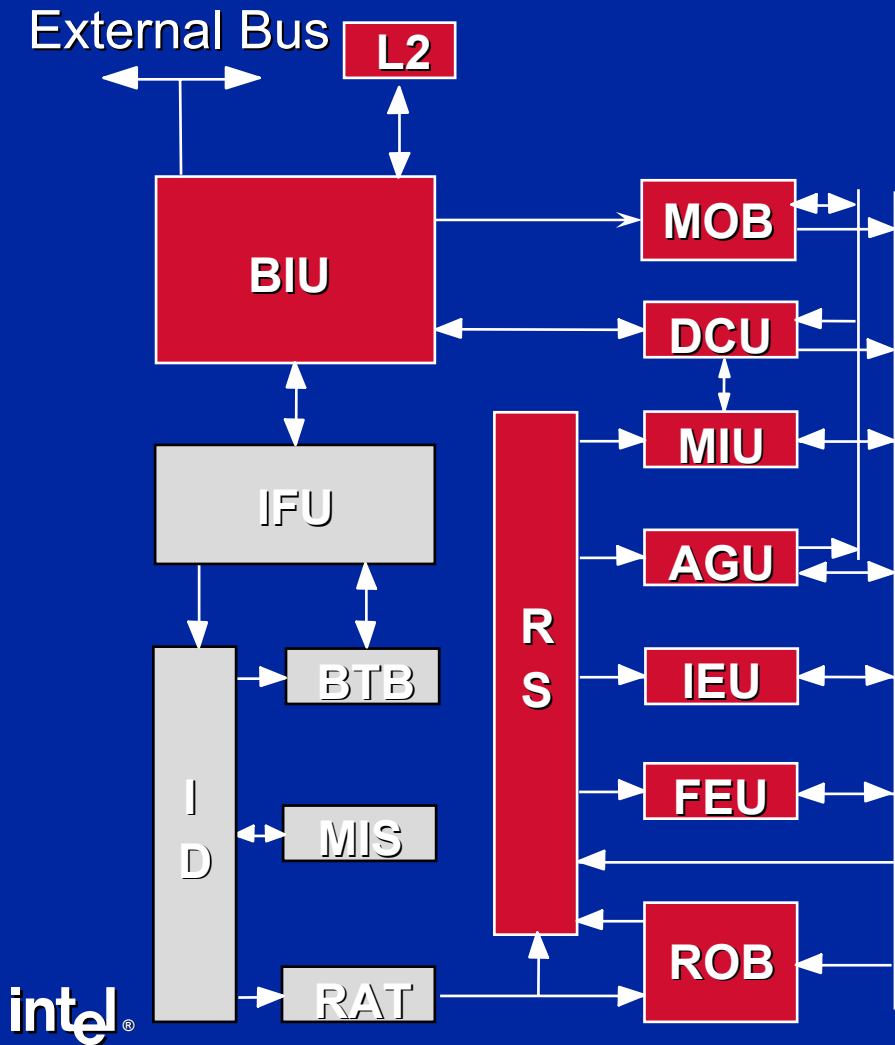
- Determine where each IA instruction starts
- Direct the bytes of each inst. to the decoder
- Convert instructions into uops.
- Buffers up to 6 uops: Enables decoders keep working even when next pipe stages are stalled.

# Alloc & RAT

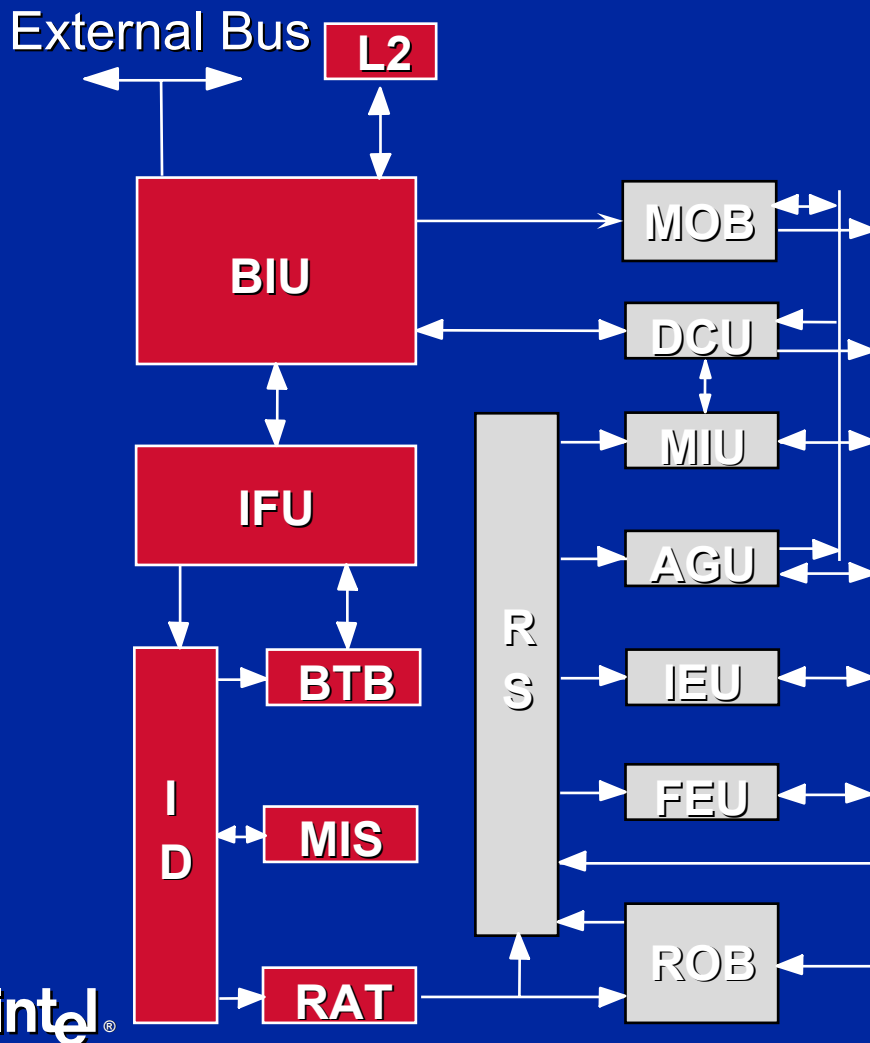
- The Allocator (Alloc) assigns each uop an entry number in the ROB
- The Register Alias Table (RAT) maps the 8 IA architectural registers into the physical registers
- Work together to perform the register allocation and renaming
- Are able to work on up to 3 uops per clock
- The Alloc also allocates Load & Store buffers in the MOB
- Special treatment for FP stack renaming

# Microarchitecture

- In-Order Front End



# Microarchitecture



## ● Out-of-order Core

- ROB: Mechanism for renaming and retirement
  - 40 entries that hold instructions in-order.
- RS: pool of all “not yet executed” instructions (up to 20)
- Execution Units
  - IEU: Integer Execution Unit
  - FEU: Floating-point Execution Unit
- Memory related units
  - AGU: Address Generation Unit MIU: Memory Interface Unit
  - DCU: Data Cache Unit
  - MOB: Orders Memory operations
  - L2: Level 2 cache

# Re-order Buffer (ROB)

- Mechanism for renaming and retirement
- Basic ROB functions
  - Provide large physical register space for register renaming
  - Buffer results of speculative execution in a 40 entry physical register file
  - Commit architectural state only after speculation (branch, exception) has resolved
  - Detect exceptions and mispredictions and initiate repair to get machine back on right track
  - Holds also the “Real Register File” - RRF

# Re-order Buffer (ROB) - cont

- **Uop flow through the ROB**
  - Uops are entered in order
  - Registers renamed by the entry#
  - Once assigned: execution order unimportant
  - After execution: entries marked “*executed*” and wait for retirement
  - An executed entry can be “*retired*” once all prior instruction have retired. That is:
    - Update “*real registers*” with value of renamed registers
    - Update memory
    - Leave the ROB
- **Only 2 read ports (for up to 6 operands)**

# Reservation station (RS)

- **Pool of all “not yet executed” uops (up to 20)**
  - Holds the uop attributes and the uop source data
- **Maintains operands status “*ready/not-ready*”**
  - Each cycle, executed uops make more operands “*ready*”
  - Uops whose all operands are *ready* can be *dispatched* for execution
  - Dispatcher chooses which of the *ready* uops to execute next
- **Responsible for:**
  - Holding the uop till it is dispatched
  - Monitoring the WB bus to capture data needed by awaiting uop
  - Bypass control of data from WB bus directly to execution unit
  - Schedule the next uops
  - Dispatch uops to functional units
  - Arbitrate the WB busses between the units

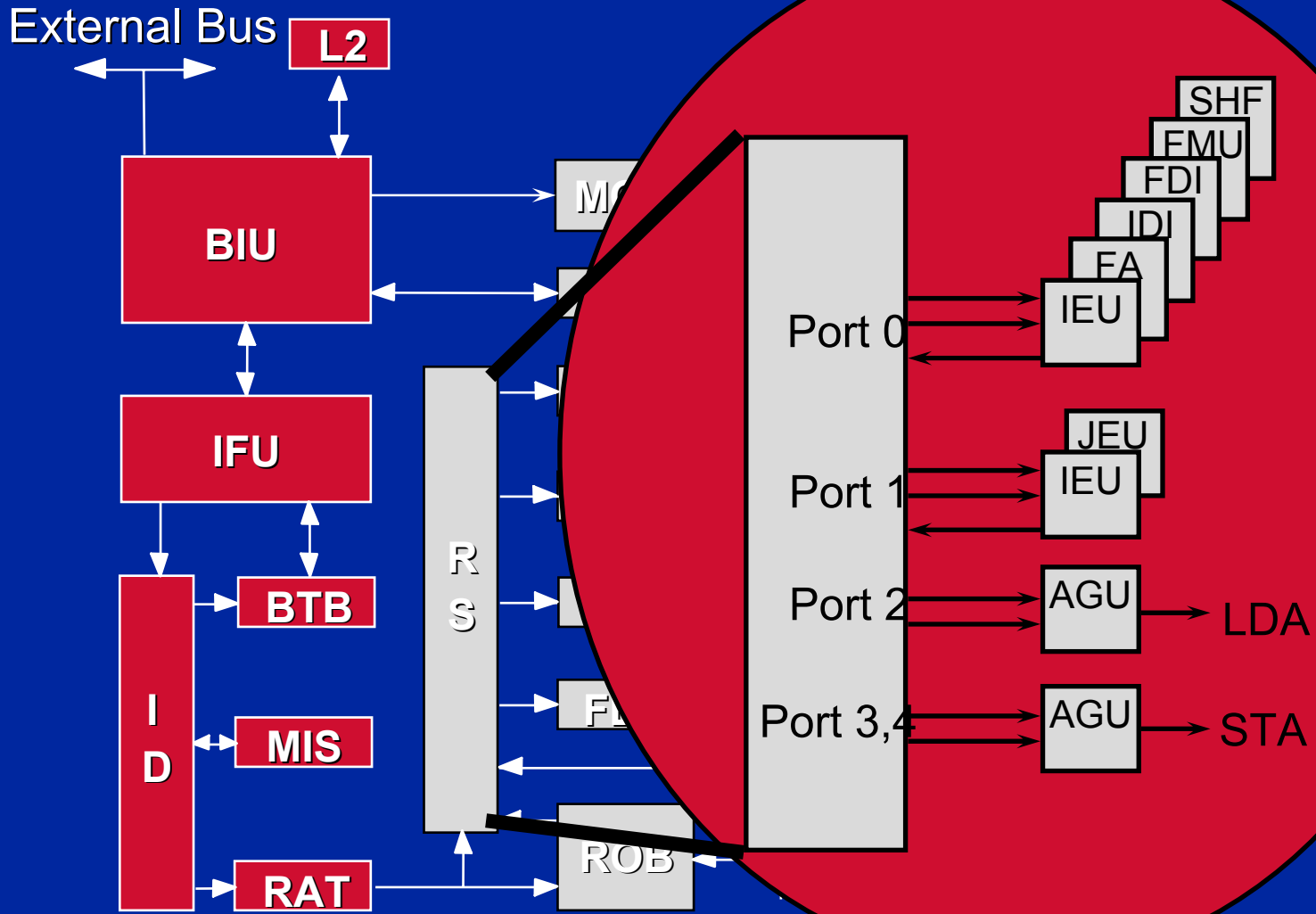
# Memory Order Buffer (MOB)

- Goal - allow out-of-order among memory operations
- Problem- Memory dependencies cannot be fully resolved statically (memory disambiguation)
  - store r1,a; load r2,b  $\Rightarrow$  can advance load before store
  - store r1,[r3]; load r2,b  $\Rightarrow$  load should wait till r3 is known
- Structure similar in concept to ROB
- Every memory uop is allocated an entry in order.
- Address & data (for stores), are updated when known
- Loads may pass loads/stores
- Stores are in order

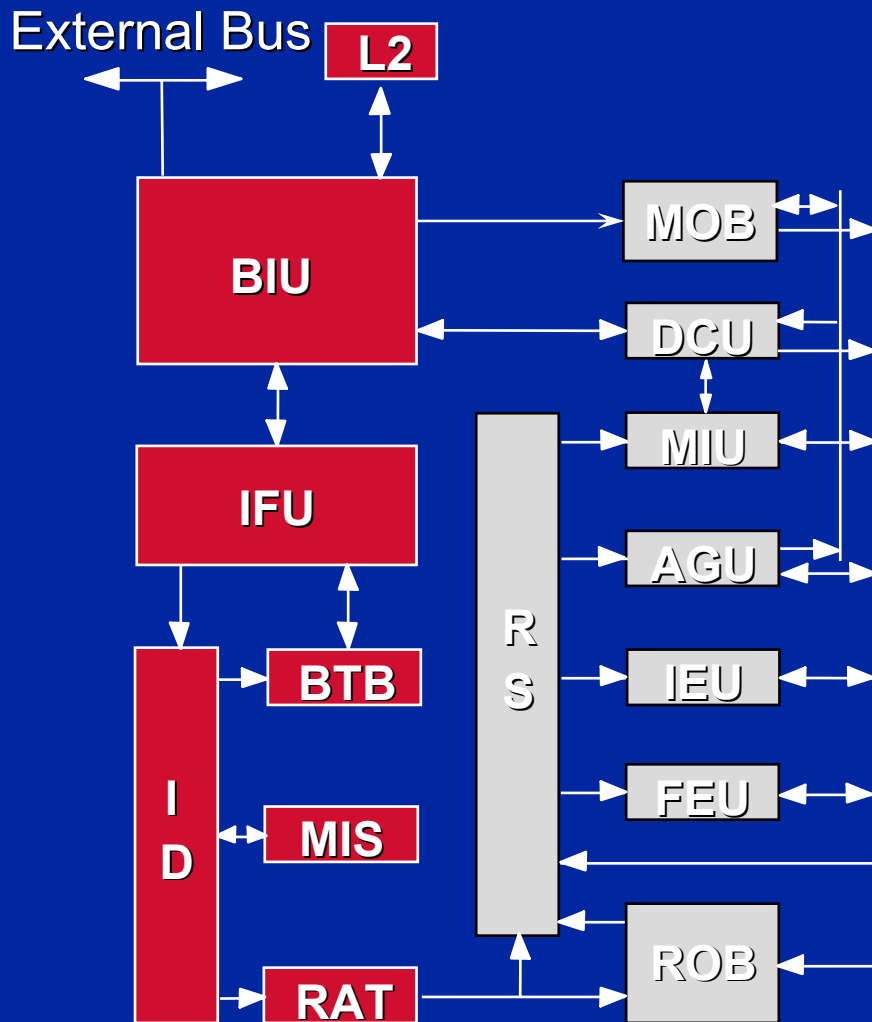
# Memory Order Buffer (MOB)

- **Load is checked against all previous stores:**
  - **Waits if store to same address exist, but data not ready**
  - **If store data exists, just use it**
  - **Waits till all previous store addresses are resolved**
  - **In case of no address collision - go to memory**

# Microarchitecture

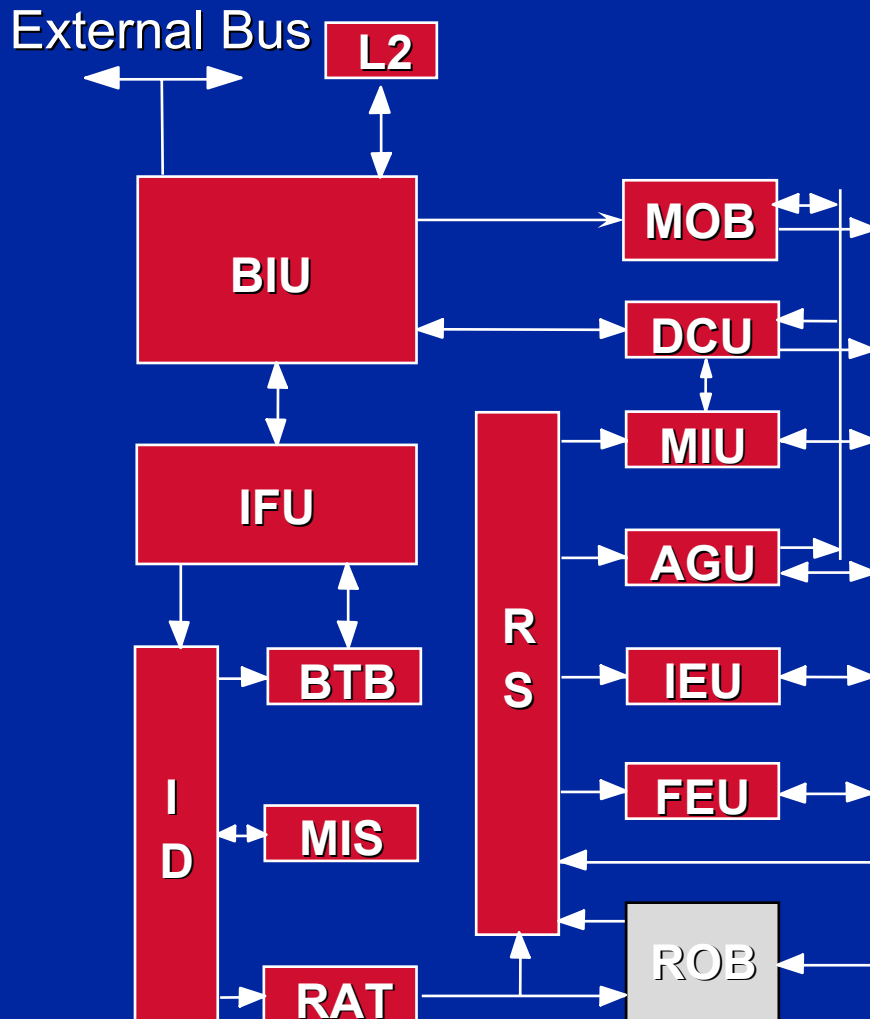


# Microarchitecture



● Out-of-order Core

# Microarchitecture



● In-Order Retire

# In order Retirement

- The process of committing the results to the architectural state of the processor
- Retires up to 3 uops per clock
- Copies the values to the RRF
- Retirement is done In Order
- Performs exception checking
- An instruction is retired after the following checks
  - Instruction has executed
  - All previous instructions have retired
  - Instruction isn't mis-predicted
  - no exceptions

# Flow of Uops through OOO Cluster

- **ISSUE:**
  - **ALLOC** unit allocates one entry per uop in the RS and in the ROB (for up to 3 uops per cycle)
    - If source data is available from the ROB (either from the RRF or from the Result Buffer (RB) it is written in the RS entry
    - Otherwise, it is marked invalid in the RS (and should be captured from the WB bus)
- **READY/SCHEDULE:**
  - Data-ready uops are checked to see if desired functional unit available
  - Up to 5 resource-ready uops are selected, and dispatched per clock
- **DISPATCH:**
  - Ship scheduled uops to appropriate functional unit (RS)
- **WRITEBACK:**
  - Capture results returned by the functional units in a result buffer (ROB)
  - Snoop result writeback ports for results that are sources to uops in RS
  - Update data-ready status of these uops (RS)

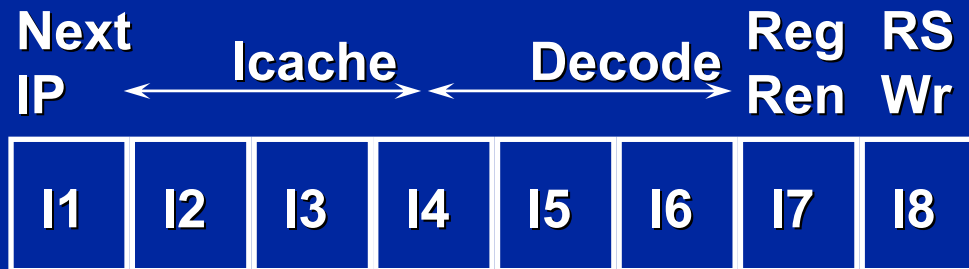
# Flow of Uops through OOO Cluster (cont)

- **RETIREMENT:**
  - 3 consecutive entries read out of the ROB
    - these entries are candidates for retirement
  - Algorithm to determine fitness for retirement: candidate is retired
    - its ready bit is set
    - it will not cause an exception
    - all preceding candidates are eligible for retirement
  - Commit results from result buffer to architecturally visible state in original “Issue” order
  - Clear machine and restart execution if “badness” occurs (ROB)

# Jump Misprediction

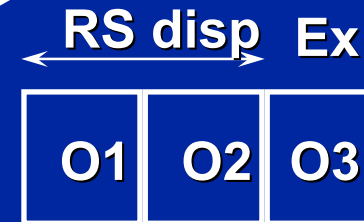
- **When the JEU detects jump misprediction it**
  - Flushes the in-order front-end and starts fetching and decoding from the “correct” execution path
  - The “correct” path may not be correct if a preceding uop that hasn’t executed yet could cause an exception
  - Therefore, the “correct” instruction stream is stalled at the RAT until the mispredicted branch retires
  - Does not flush the OOO part of the machine, since all instructions preceding the jump must be completed and retired
- **When the mispredicted branch retires from the ROB**
  - Resets all state in the Out-of-Order Engine (RS, RB, MOB, etc.)
  - Un-stalls the in-order machine
  - The RS immediately grabs the correct uops from the RAT and starts scheduling and dispatching them

# Pipeline

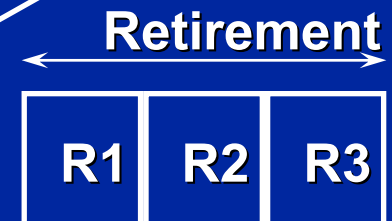


• In-Order Front End

• Out-of-order Core



• In-order Retirement



- 1: Next IP
- 2: ICache lookup
- 3: IC2 /ILD (instruction length decode)
- 4: IC3/rotate
- 5: ID1
- 6: ID2
- 7: RAT- rename sources,  
ALLOc-assign destinations
- 8: ROB-read sources  
RS-schedule data-ready uops for dispatch
- 9: RS-dispatch uops
- 10: EX
- 11: Retirement

# OOO Concept Example

- Lets follow this code:

<u>program counter</u>	<u>Instruction</u>	<i>[format: op src, dest]</i>	
n	mov	r4,r1	Cycle 1 decode
n+1	add	r1,r2	
n+2	mov	M2,r1	
n+3	add	r1,r3	
n+4	jmp	L2	Cycle 2 decode
n+5	add	r3,r4	
n+6	mov	M3,r1	
n+7	add	r1,r4	
n+8	dec	r5	Cycle 3 decode

- Every cycle, 4 instructions are decoded

intel® *A demonstrating example, where RS and Rob are combined*

# Code Example (rename & Sched)

Lets follow this code:

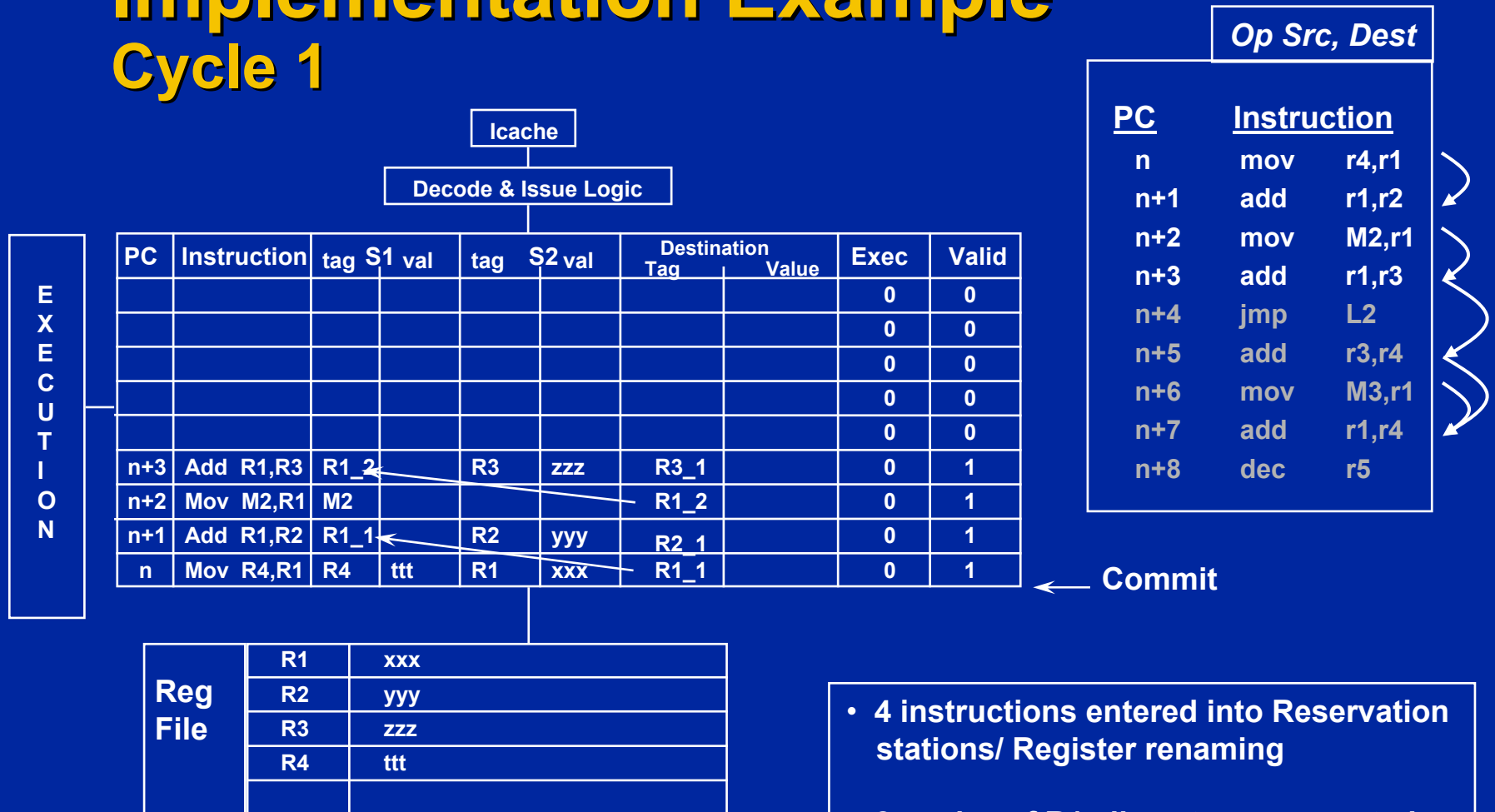
<u>PC</u>	<u>Instructions</u>	<u>After Renaming</u>	<u>Execution</u>
n	mov r4,r1	r4, r1_1	D E W
n+1	add r1,r2	r1_1, r2, r2_1	D E W
n+2	mov M2,r1	M2, r1_2	D E W
n+3	add r1,r3	r1_2, r3, r3_1	D E W
n+4	jmp L2	...	D E W
n+5	add r3,r4	r3_1, r4, r4_1	D E W
n+6	mov M3,r1	M3, r1_3	D E W
n+7	add r1,r4	r1_3, r4_1, r4_2	D E W
n+8	dec r5	r5, r5_1	D E W

*cycle:* 0 1 2 3 4 5 6

- Every cycle, 4 instructions are decoded

# Implementation Example

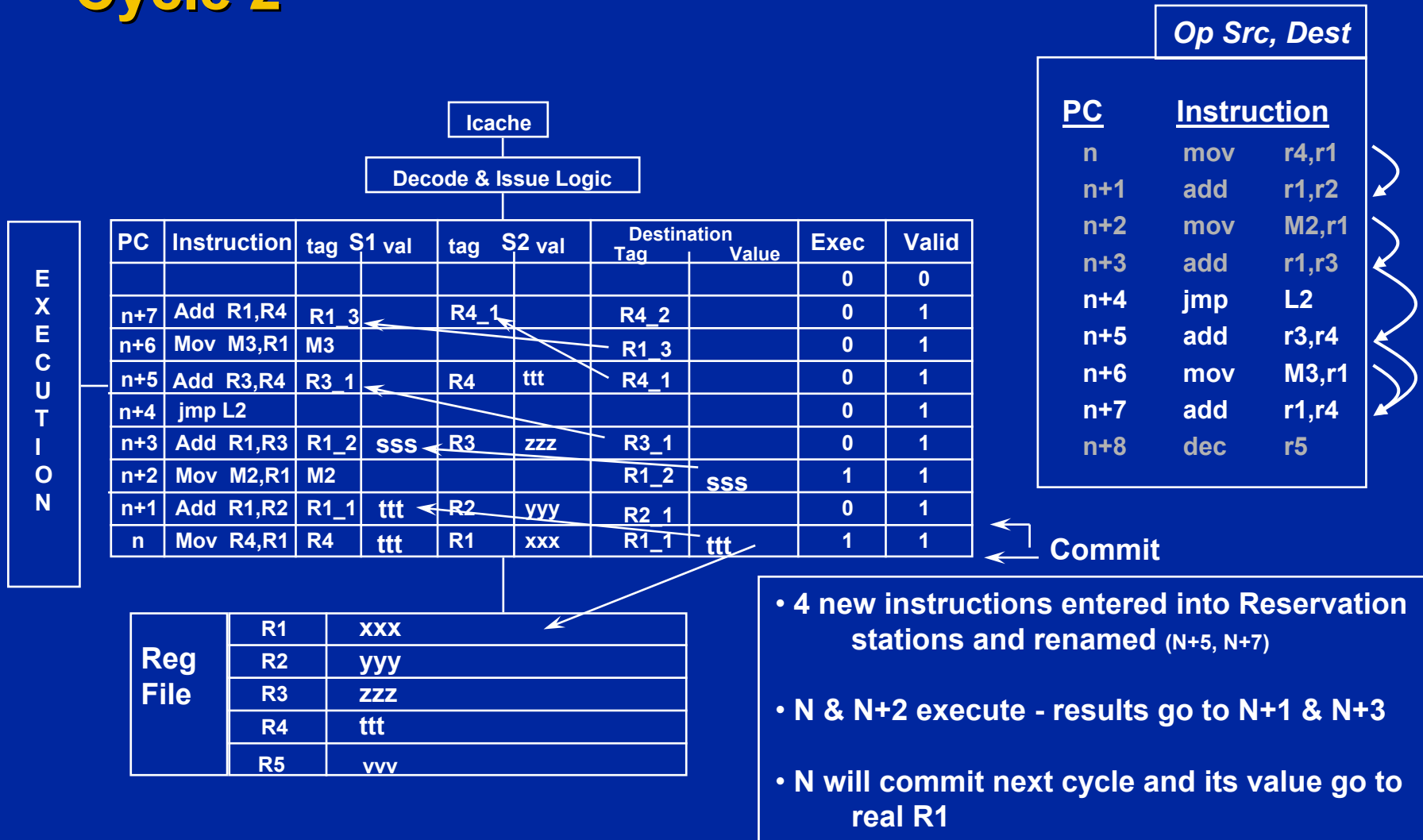
## Cycle 1



- 4 instructions entered into Reservation stations/ Register renaming
- 2 copies of R1 alive - tags connected
- Available values read from committed reg file

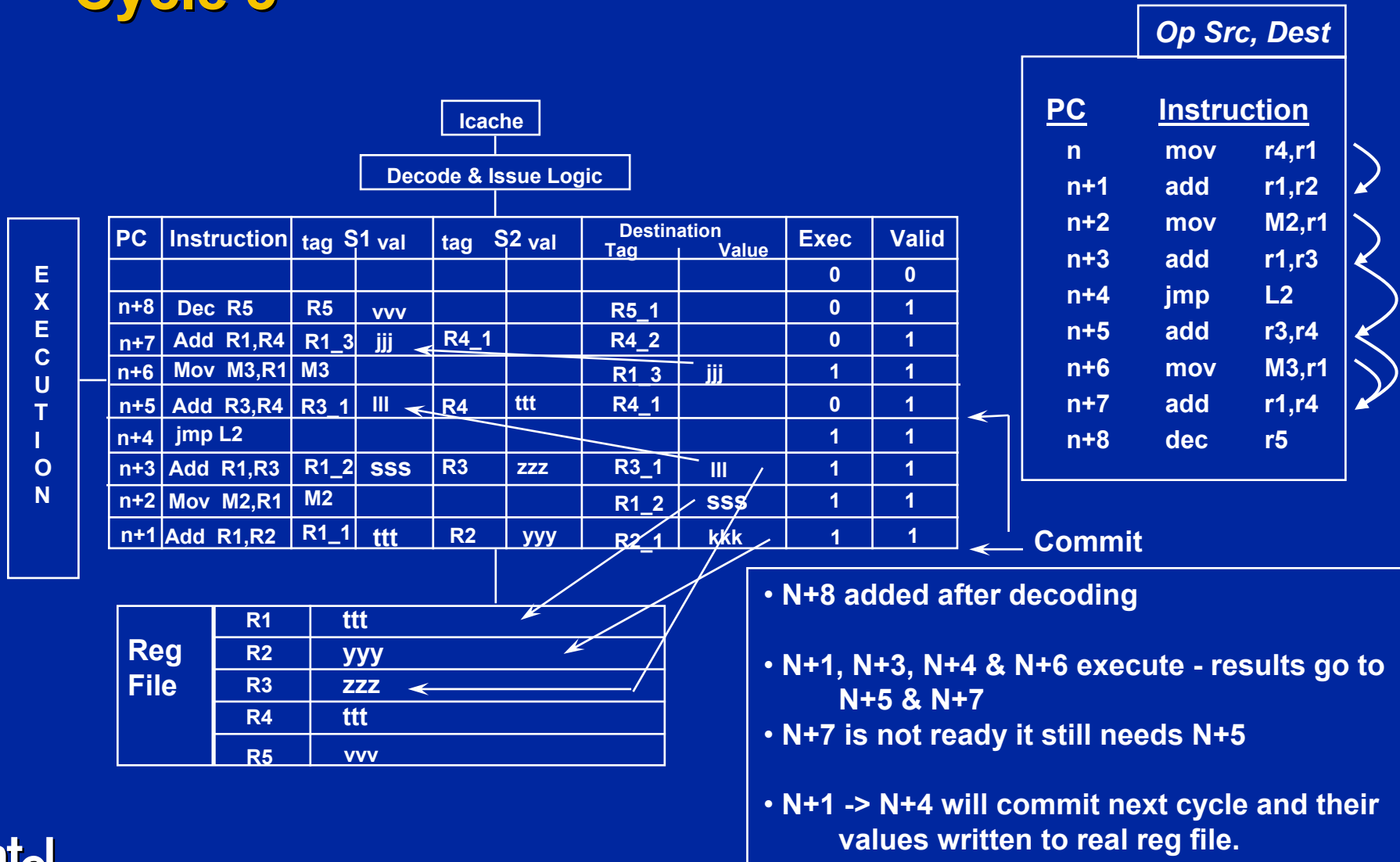
# Implementation Example (Cont...)

## Cycle 2



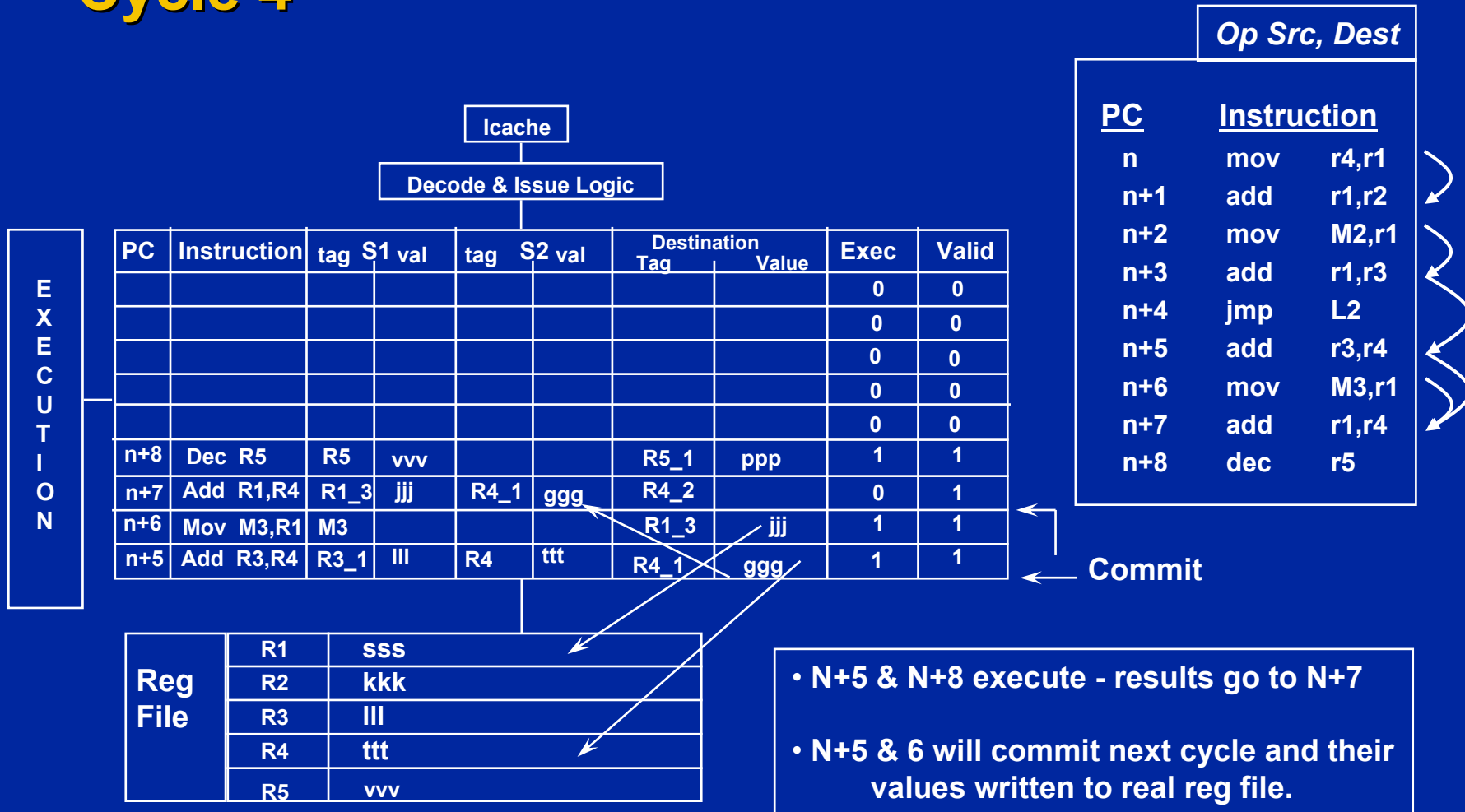
# Implementation Example (Cont...)

## Cycle 3



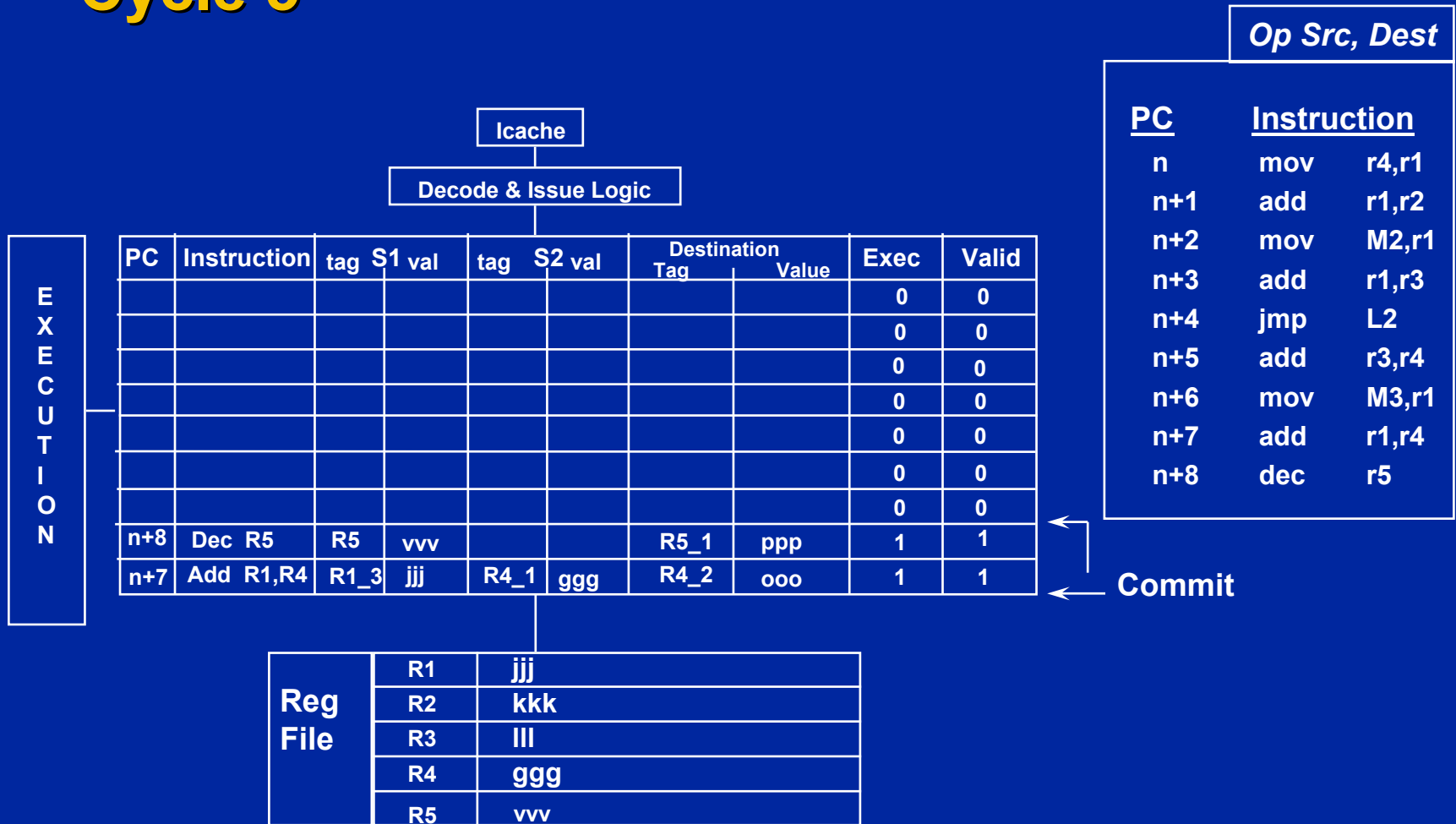
# Implementation Example (Cont...)

## Cycle 4



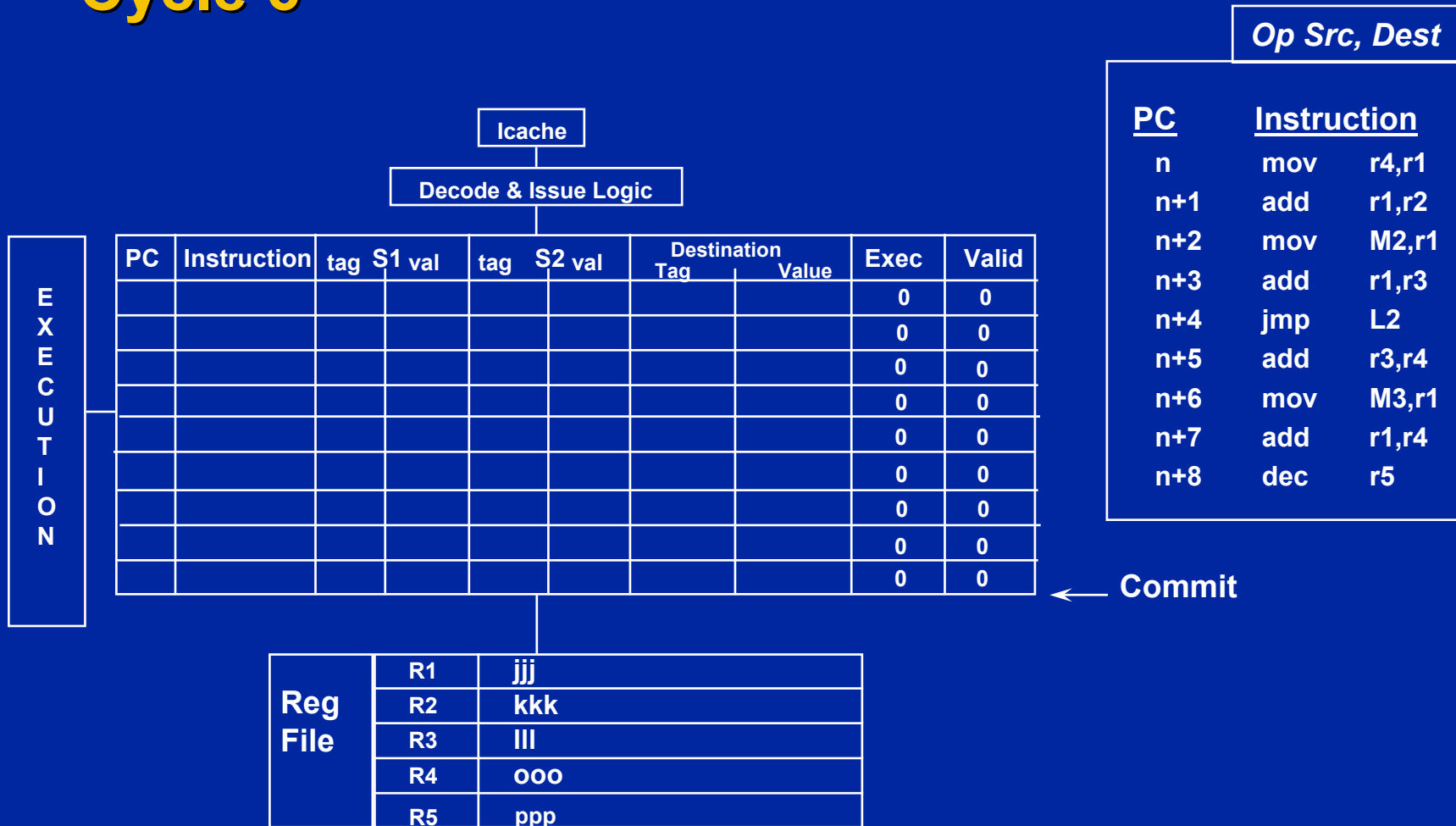
# Implementation Example (Cont...)

## Cycle 5

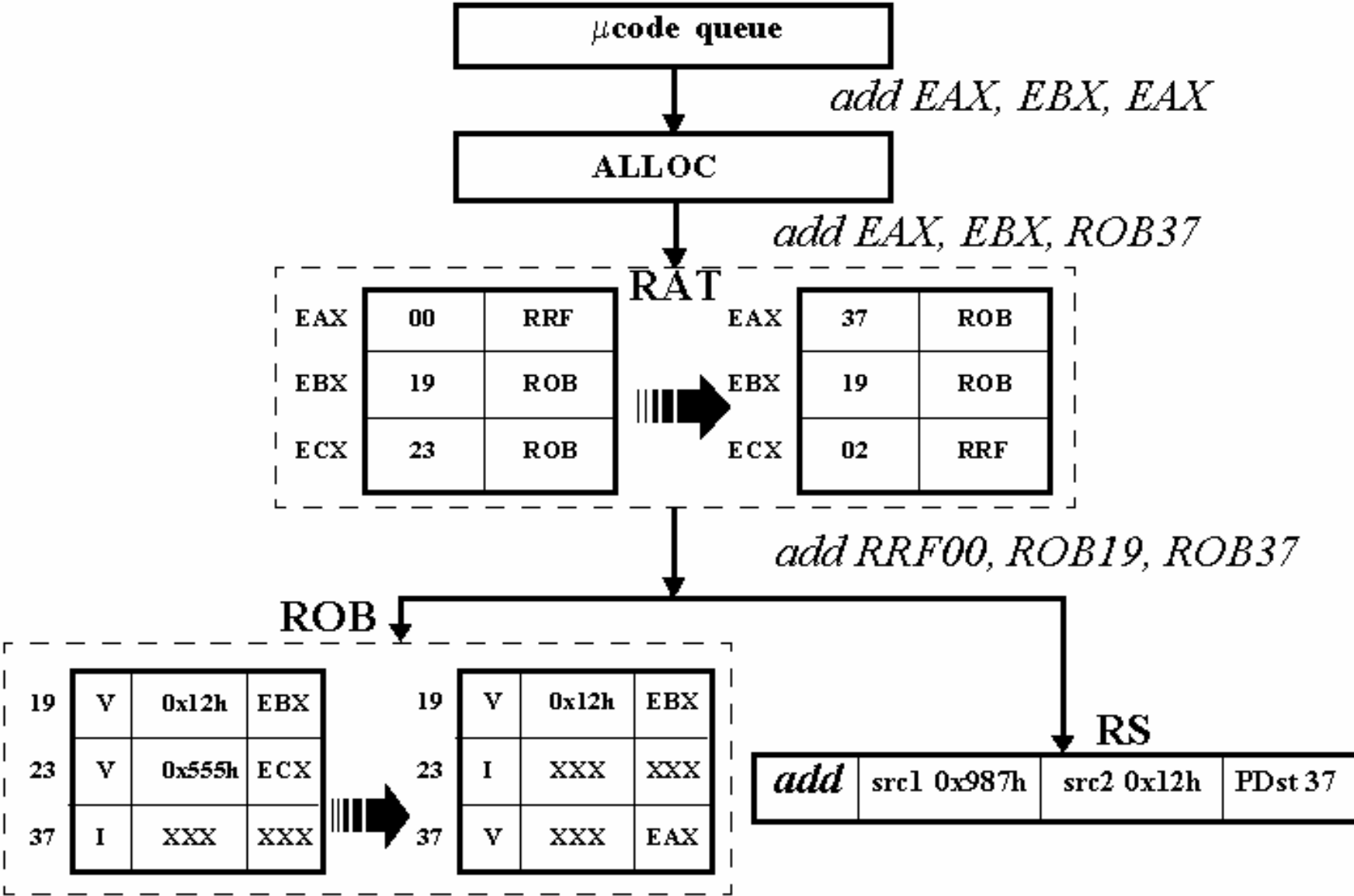


# Implementation Example (Cont...)

## Cycle 6



# Register Renaming example

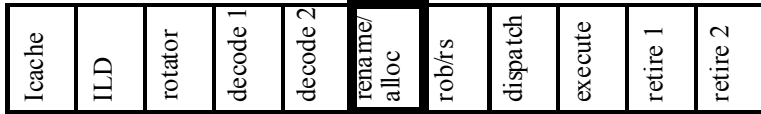


# An OOOE Example - Issue

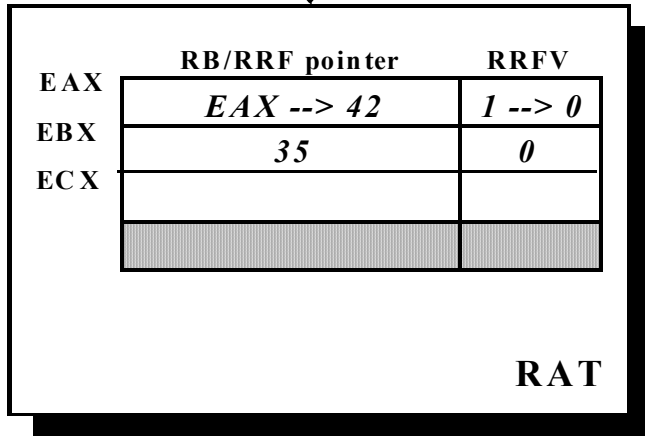
*add EAX, EBX → EAX*

*sub EAX, 414 → ECX*

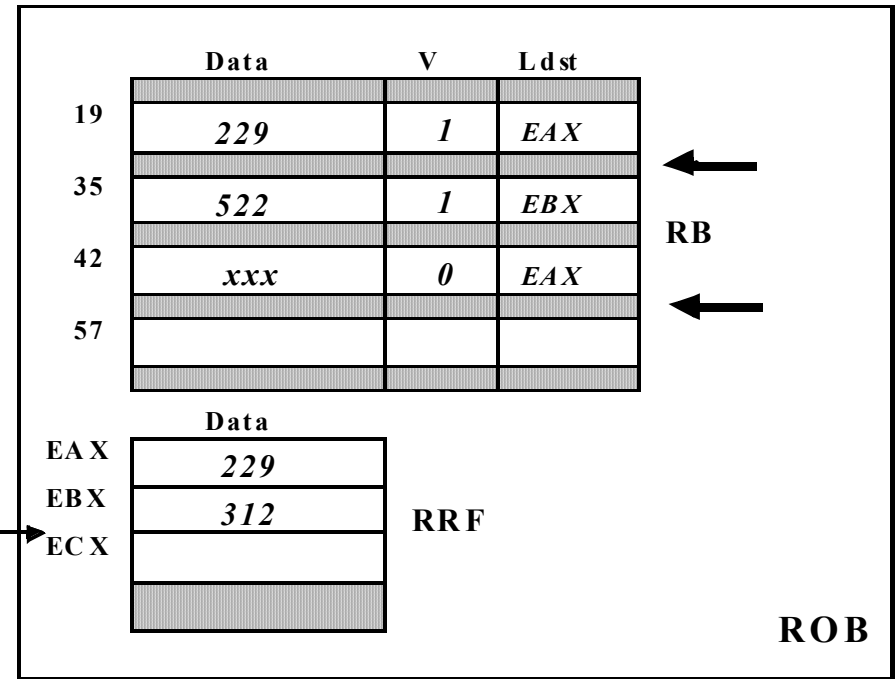
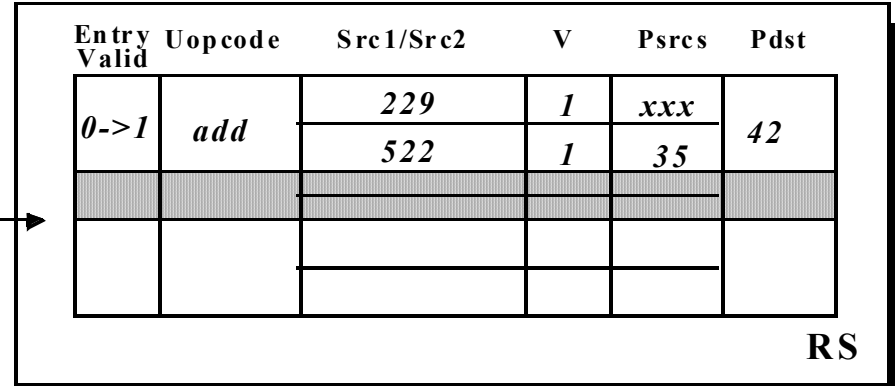
## ISSUE



*add EAX, EBX, EAX*



*add xxx, 35, 42*  
*EAX, EBX, EAX*  
*RRFV*

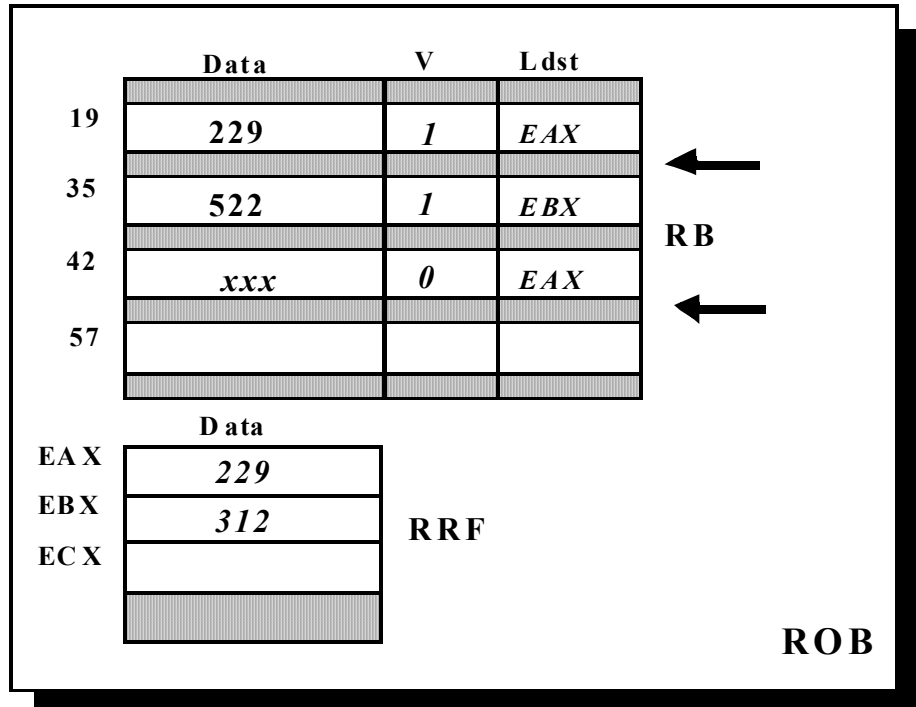
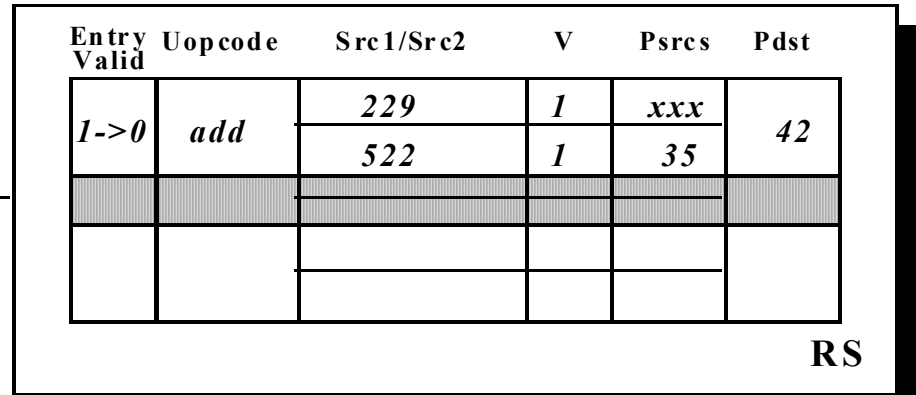
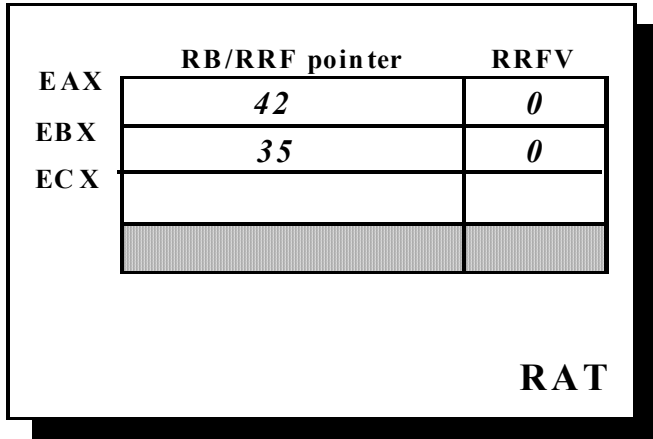
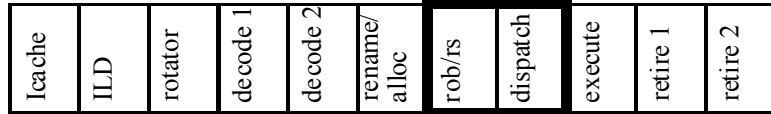


# An OOOE Example - Dispatch

## READY/SCHEDULE/DISPATCH

*add EAX, EBX → EAX*  
*sub EAX, 414 → ECX*

*add 229, 522, pdst=42*

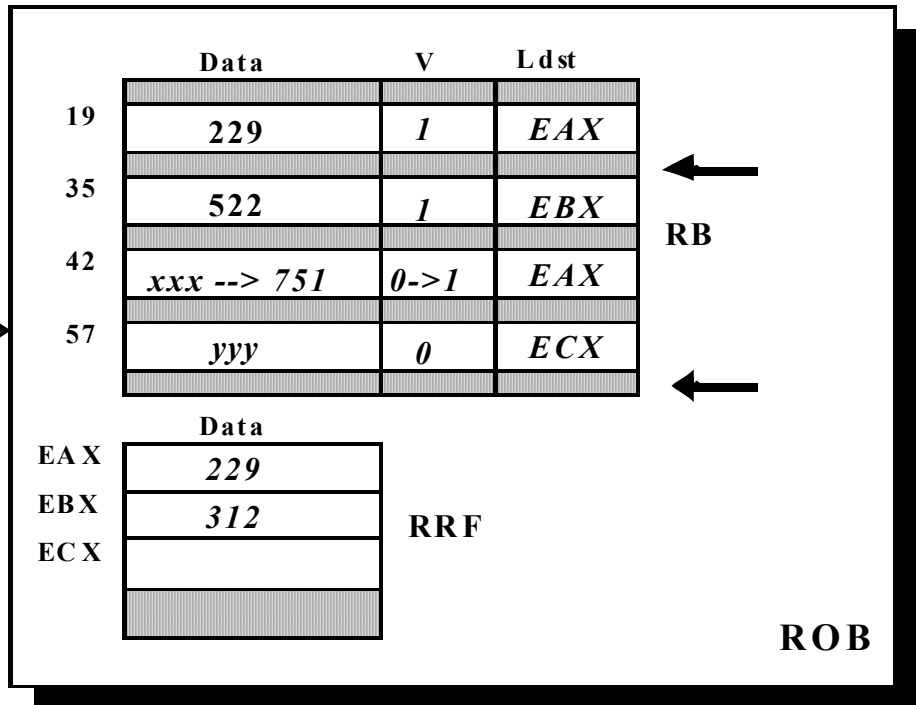
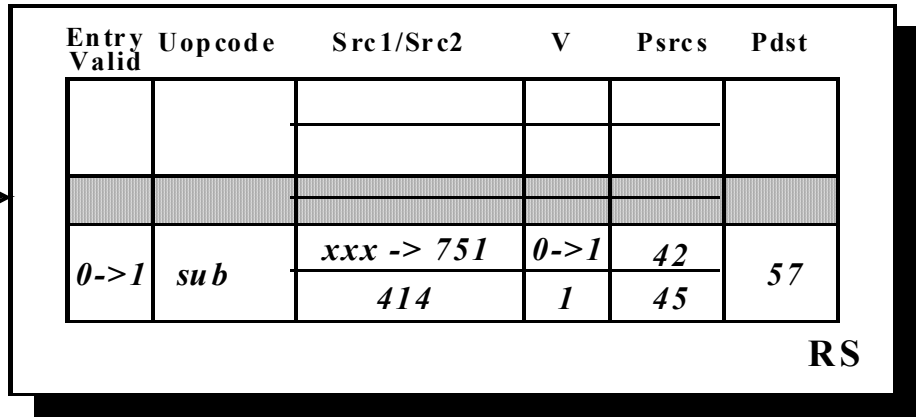
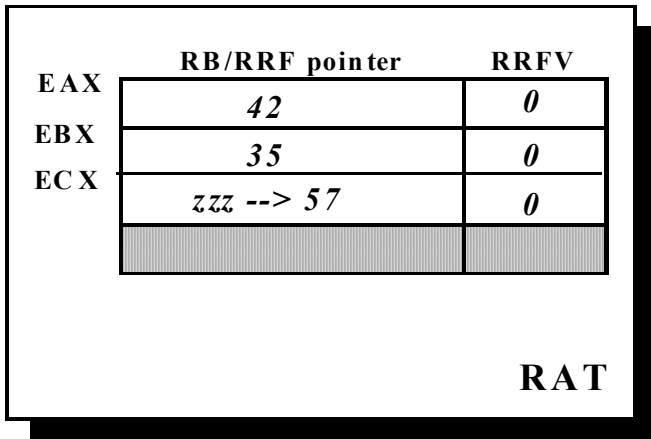
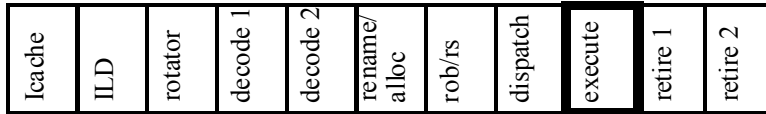


# An OOOE Example - Execute

## WRITEBACK

*add EAX, EBX → EAX*  
*sub EAX, 414 → ECX*

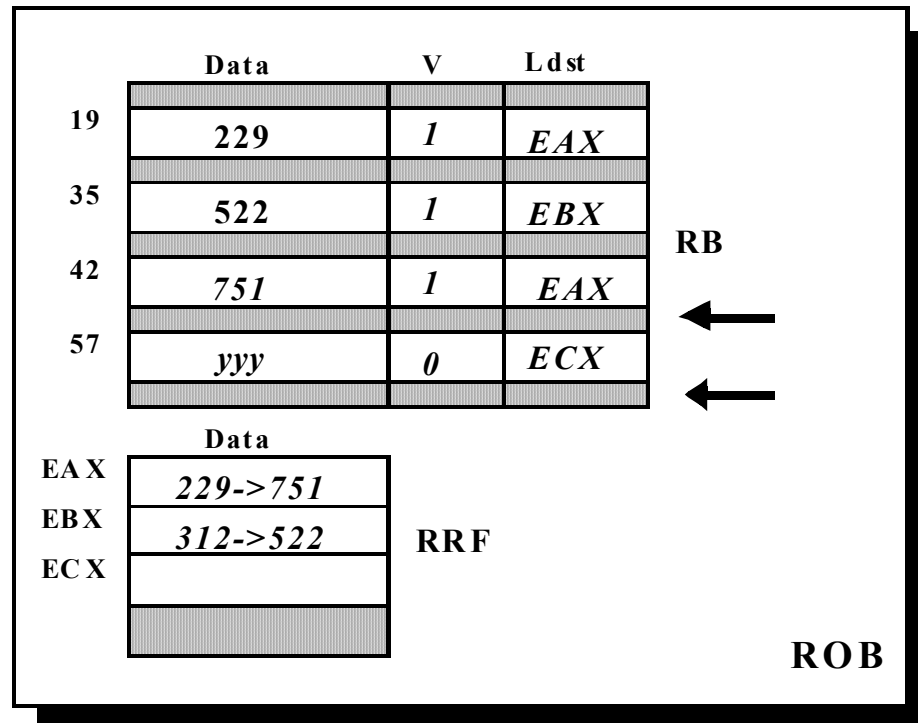
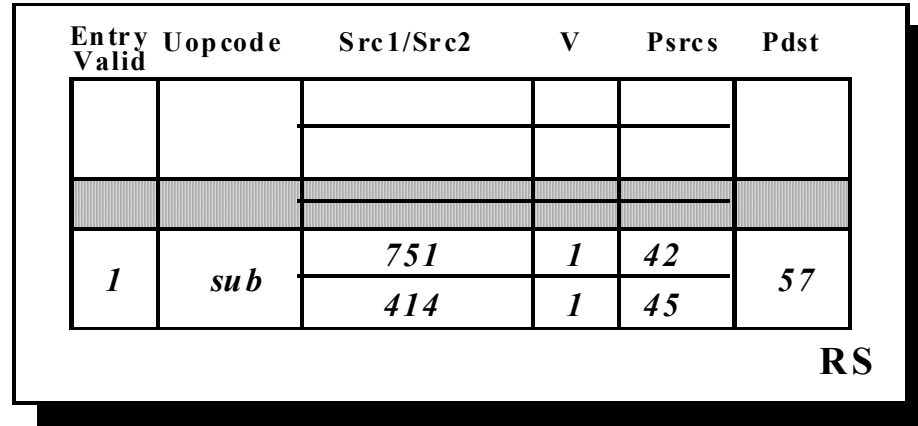
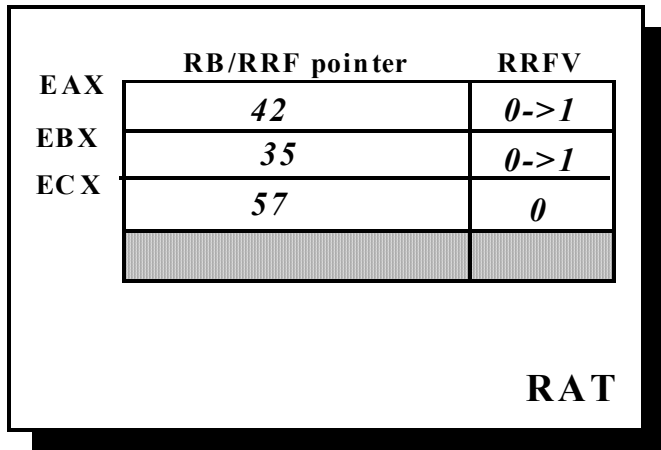
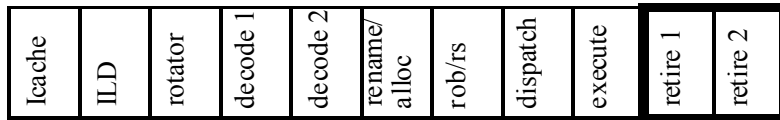
*result=751, pdst=42*



# An OOOE Example - Retire

## RETIREMENT

*add EAX, EBX* → *EAX*  
*sub EAX, 414* → *ECX*



**Hope you liked that...**

# SW Aspects

- “Flaky” (unpredictable) branches
- Partial Stalls
- Data mis-alignment

# Pentium-M Processor (Baniyas)

- **Most significant, major improvement over P6 architecture ever**
- **Key  $\mu$ arch changes**
  - New, improved branch prediction
    - BLG, loop predictor, indirect branch predictor
  - “Stack manager” – reduce >5% of uops
  - Uop fusion – reduce >10% of uops
  - P4 bus, bigger caches
- **A lot of power-aware techniques all over**