Computer Architecture

The P6 Microarchitecture

An Example of an Out-Of-Order Micro-processor

By Dan Tsafrir, 2/5/2011, 16/5/2011
Presentation based on slides by Lihu Rappoport and Adi Yoaz

The P6 Family (i686)

- **Features**
  - 1st out of order x86 (=> data flow analysis)
  - Speculative execution (across branches; requires flush+recovery)
  - Multiple branch prediction (wide op window contains 5 branch on avg)
  - Register renaming (solves false dependencies, gives more regs)
  - Super-pipeline: ~12 pipe stages (P-IV had 31! i7 back to 14)

Processor	Year	Freq (MHz)	Bus (MHz)	L2 cache	Feature size**
Pentium® Pro	1995	150~200	60/66	256/512K*	0.5, 0.35μm
Pentium® II	1997	233~450	66/100	512K*	0.35, 0.25μm
Pentium® III	1999	450~1400	100/133	256/512K	0.25, 0.18, 0.13μm
Pentium® M	2003	900~2260	400/533	1M / 2M	0.13, 90nm
Core™	2005	1660~2330	533/667	2M	65nm
Core™ 2	2006	1800~2930	800/1066	2/4/8M	65nm

*off die ** size of smallest part is smaller than the feature size

The P6 Family (i686)

- Still used:
  - MacBook Air (1.4GHz Core 2 Duo)
  - Good for low power consumption
- Clock frequency ~proportional to feature size
- After P-III came P-IV… which wasn't ideal for mobile computing
- Much (not all) of the improvement comes from feature size minimization

Processor	Year	Freq (MHz)	Bus (MHz)	L2 cache	Feature size**
Pentium® Pro	1995	150~200	60/66	256/512K*	0.5, 0.35μm
Pentium® II	1997	233~450	66/100	512K*	0.35, 0.25μm
Pentium® III	1999	450~1400	100/133	256/512K	0.25, 0.18, 0.13μm
Pentium® M	2003	900~2260	400/533	1M / 2M	0.13, 90nm
Core™	2005	1660~2330	533/667	2M	65nm
Core™ 2	2006	1800~2930	800/1066	2/4/8M	65nm

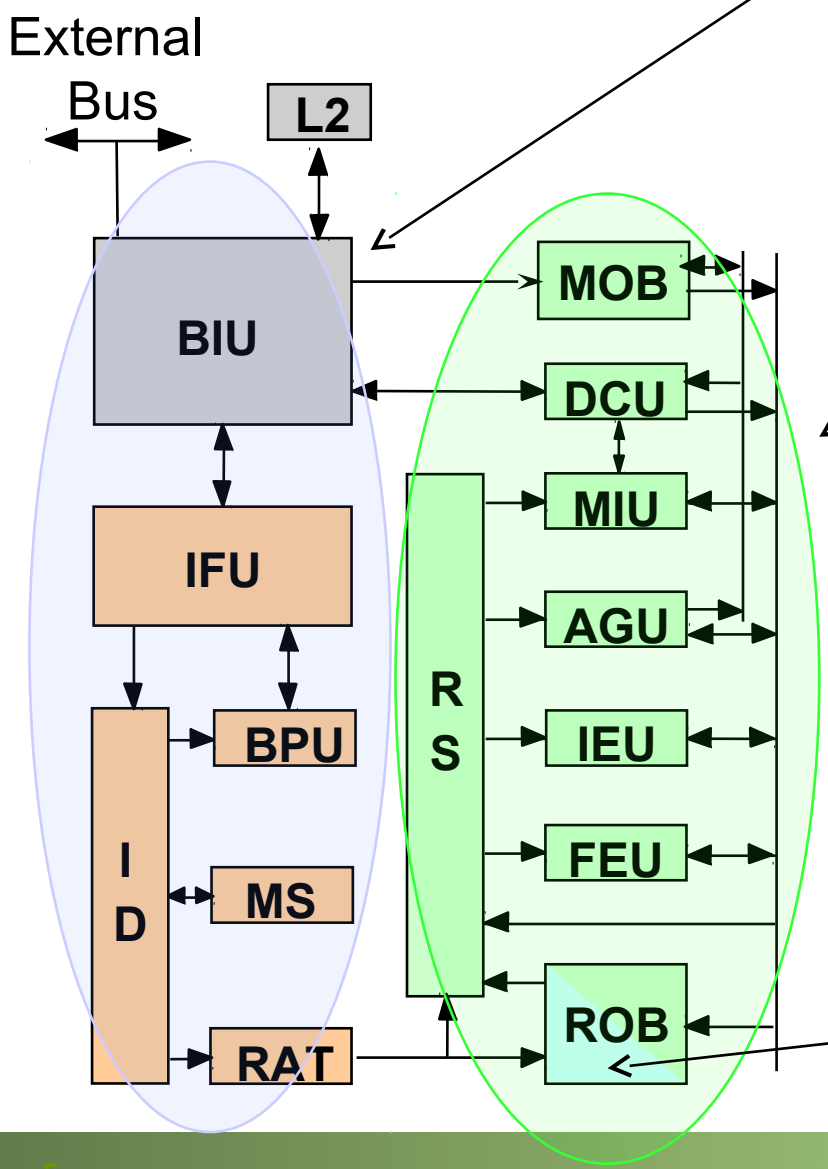
*off die

** size of smallest part is smaller than the feature size

Chip logically partitioned to 3

- **Front end**
  - In order, get and ops from memory
  - Decode them + turn them from CISC ops to >=1 u-ops
  - Uops are RISC-like
  - So x86 input=CISC, but internally it's RISC
  - The front-end is responsible to make the transition
- **Core**
  - Out of order, speculative, superscalar, renames registers
- **Retire**
  - In order
  - When speculation ends, commit
  - Can simultaneously commit <= 3 (the "width" of the machine)

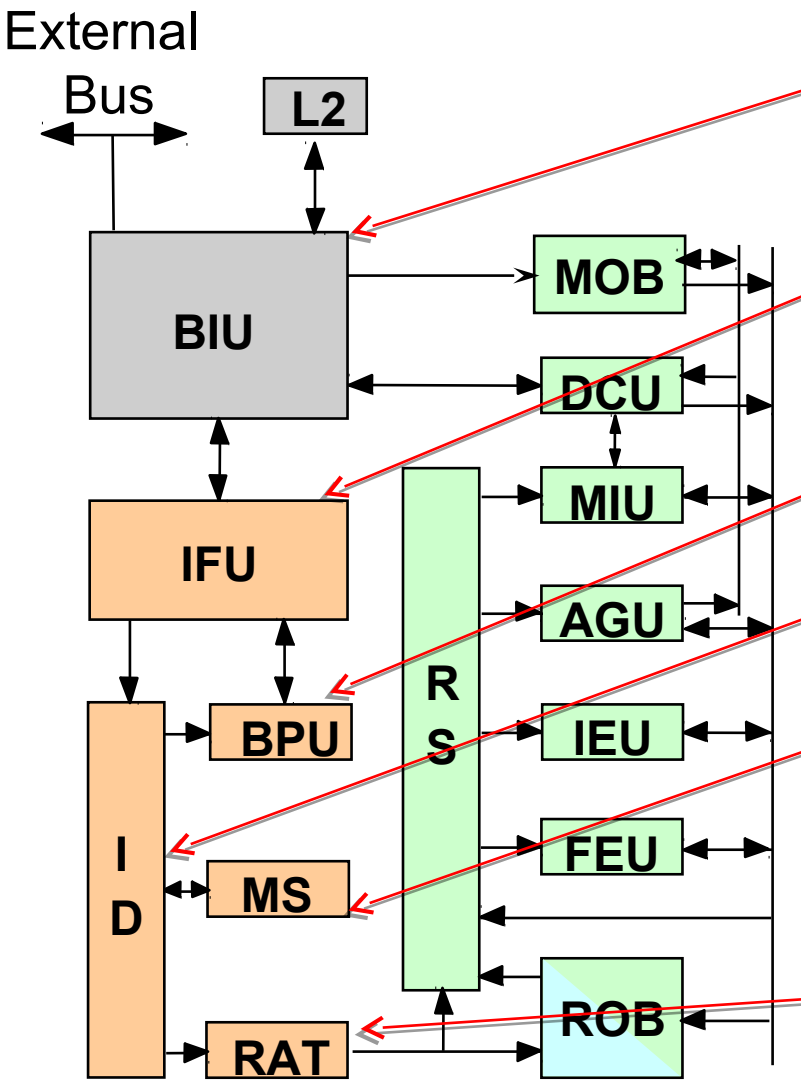
P6 µArch



- **In-Order Front End**
  - BIU: Bus Interface Unit
  - IFU: Instruction Fetch Unit (includes IC)
  - BPU: Branch Prediction Unit
  - ID: Instruction Decoder
  - MS: 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**

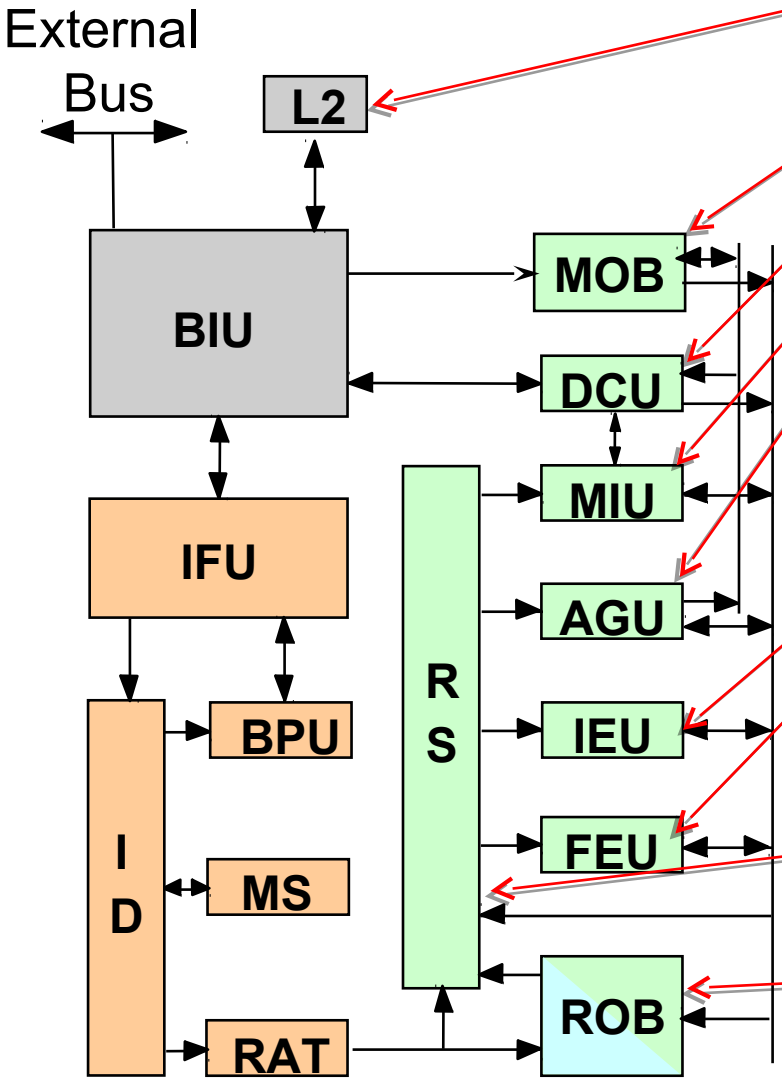
P6 µArch

In-Order Front End



- BIU: Bus Interface Unit (fetches instructions)
- IFU: Instruction Fetch Unit (includes i-cache)
- BPU: Branch Prediction Unit
- ID: Instruction Decoder
- MS: Micro-Instruction Sequencer (complex ops are comprised of a sequence of µ-ops; simple ops are comprised of only 1 µ-op)
- RAT: Register Alias Table (solves false dep.; most recent arch => physical mapping)

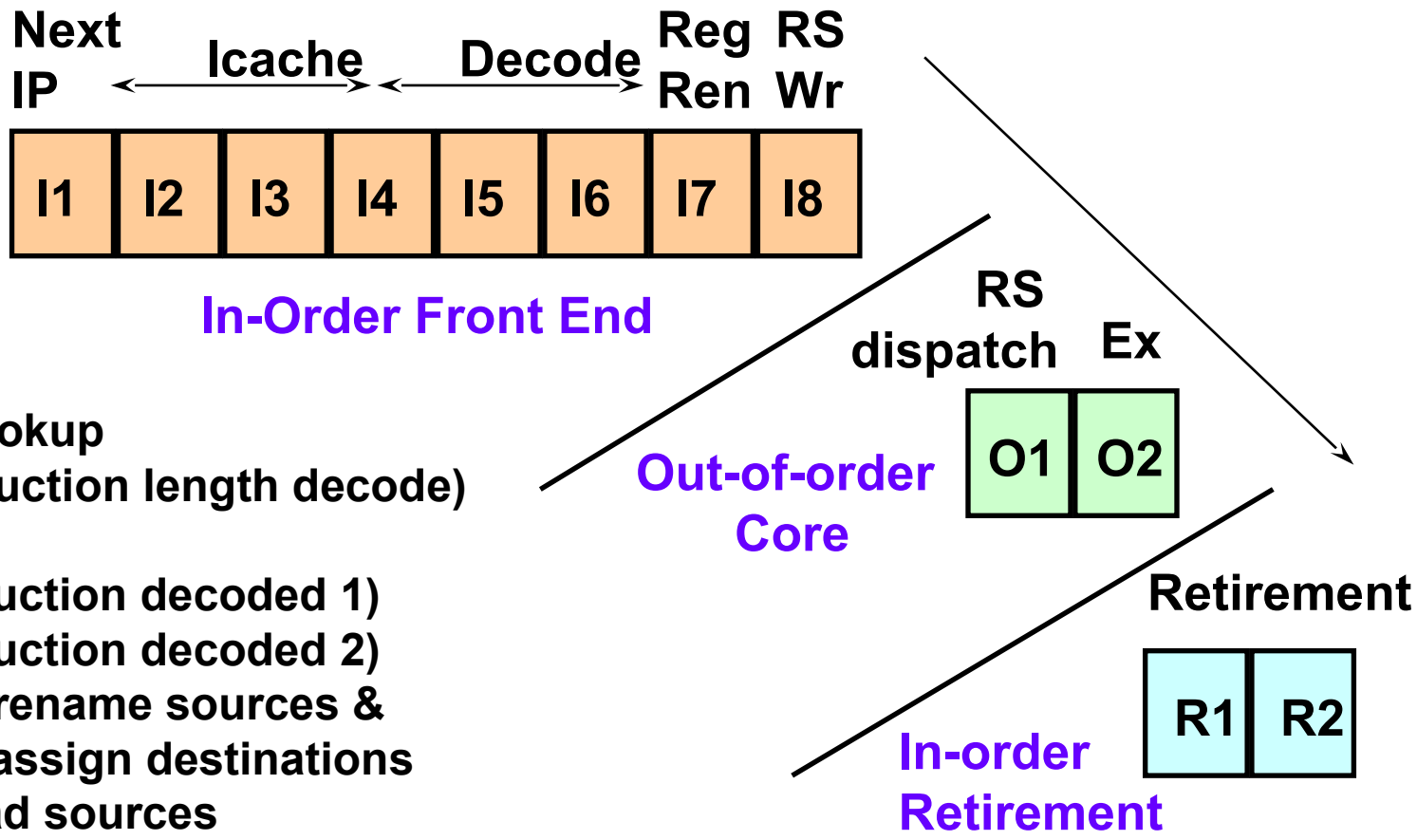
P6 µArch



Out-of-order Core

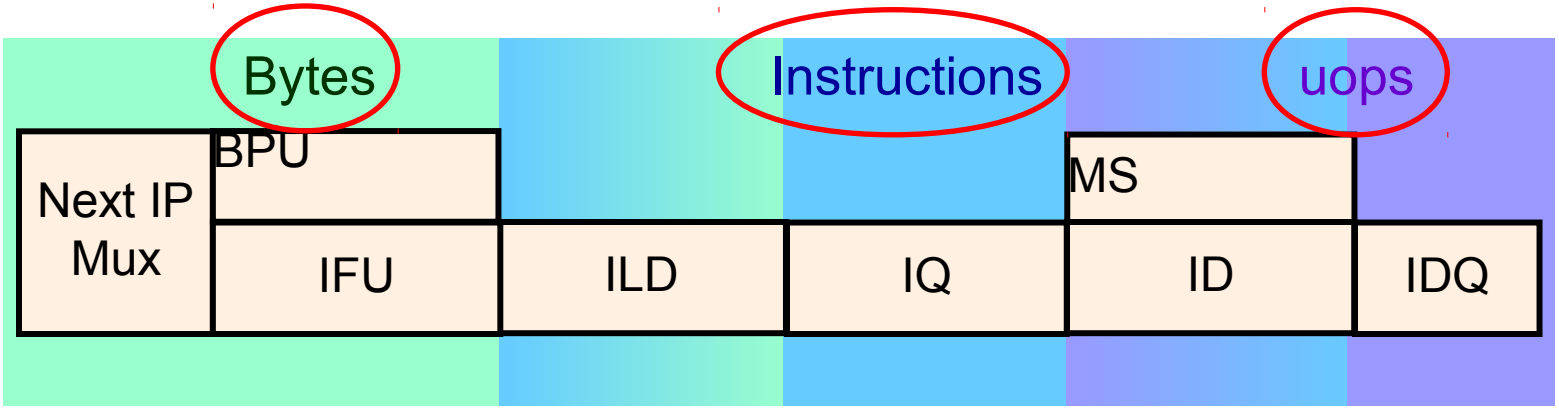
- L2: Level 2 cache
- MOB: Memory Order Buffer
- DCU: Data Cache Unit
- MIU: Memory Interface Unit
- AGU: Address Generation Unit
- RRF: "Real" Register File
  (not shown; the machine's state)
- IEU: Integer Execution Unit
- FEU: Floating-point Execution Unit
- RS: Reservation Stations
  (All those ops whose dependencies aren't yet met; up to 20; when read, 5 ports to exe units)
- ROB: Reorder Buffer
  (The physical regs; one entry per op – the reg is the dest of the op; in order!)

P6 Pipeline - 12 stages (10<=P6<=14)



1: Next IP
2: ICache lookup
3: ILD (instruction length decode)
4: rotate
5: ID1 (instruction decoded 1)
6: ID2 (instruction decoded 2)
7: RAT - rename sources &
ALLOC - assign destinations
8: ROB - read sources
RS - schedule data-ready uops for dispatch
9: RS - dispatch uops
10:EX
11-12: Retirement

In-Order Front End



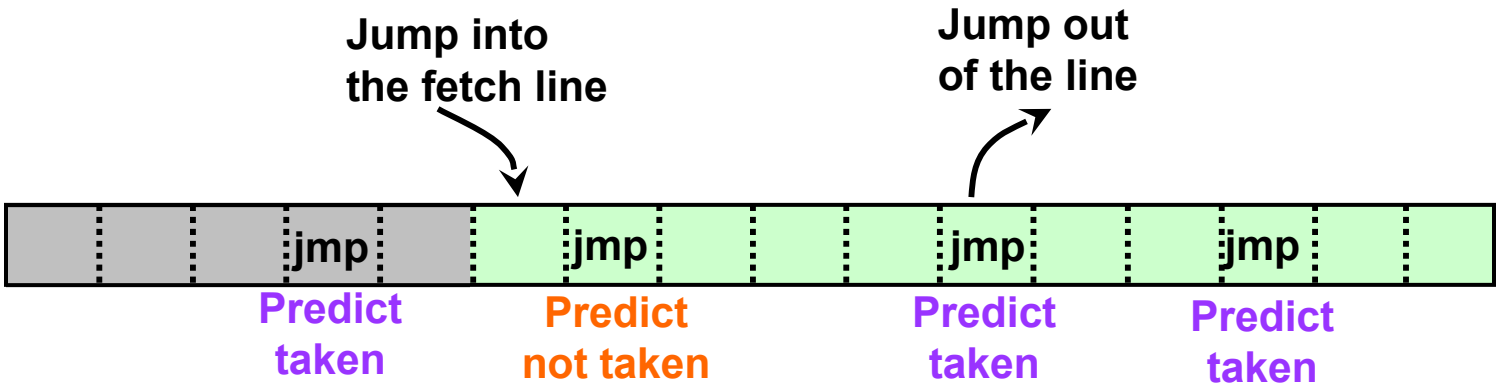
- **BPU – Branch Prediction Unit – predict next fetch address**
- **IFU – Instruction Fetch Unit**
  - iTLB translates virtual to physical address
  - IC supplies 16byte/cyc (access L2 cache, maybe memory, on miss)
- **ILD – Induction Length Decode – split bytes to instructions**
- **IQ – Instruction Queue – buffer the instructions**
- **ID – Instruction Decode – decode instructions into uops**
- **MS – Micro-Sequencer – provides uops for complex instructions**
- **IDQ – Instruction Decode Queue – buffer the uops**

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 misprediction
  - 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

Branch Prediction – Clustering

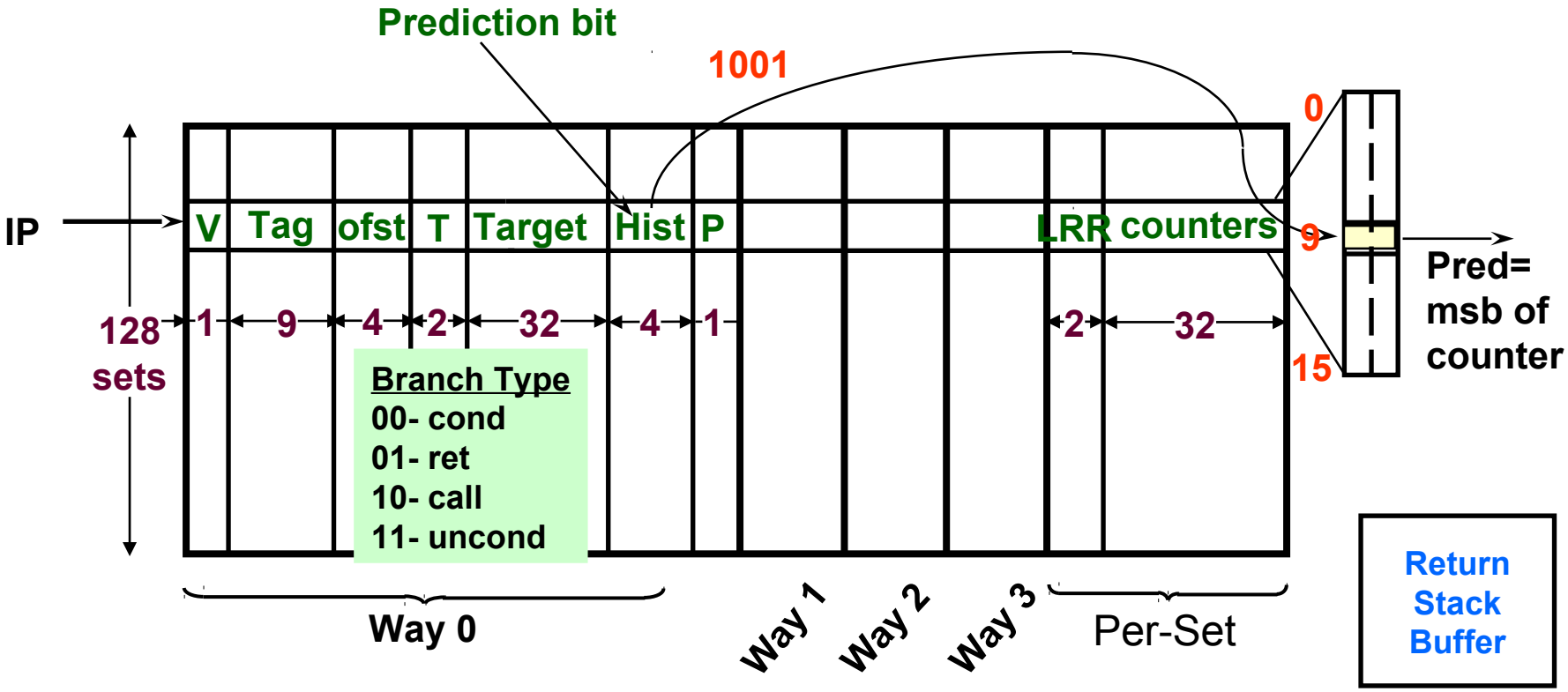
- **Given a fetched line (bytes), need to know which line to fetch next**
  - Perhaps there's more than one branch in the line
  - We must use the 1st (leftmost) taken branch (>= the fetched IP)



- **Implemented by**
  - Splitting IP into *set* + *tag* + *offset* (within line)
  - If there's a match
    - The offsets of the matching ways are ordered
    - Ways with offset smaller than the fetch IP offset are discarded
    - The 1st branch that's predicted taken is chosen as the predicted branch

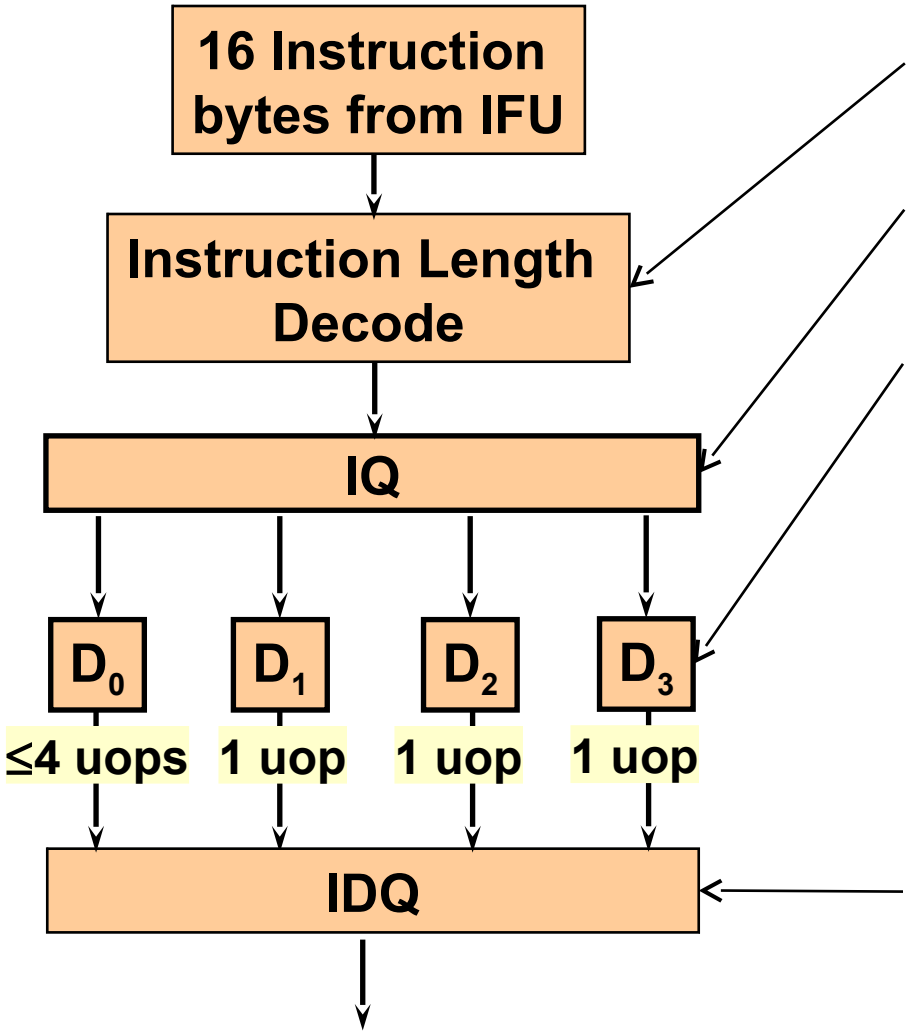
The P6 BTB

- 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

In-Order Front End: Decoder



Determine where each x86 op starts

Buffers ops in queue

Convert ops into uops

- D0 handles complex ops (translate to up to 4 uops)
- D1-3 for simple ops that translate to 1uop
- In case ops aligned with D1-3 decode into >1 uops, defer them to following cycle (whereby D0 will be used)

Buffers uops

- Smooth decoder's variable throughput (we want the same number of ops every cycle (=4))

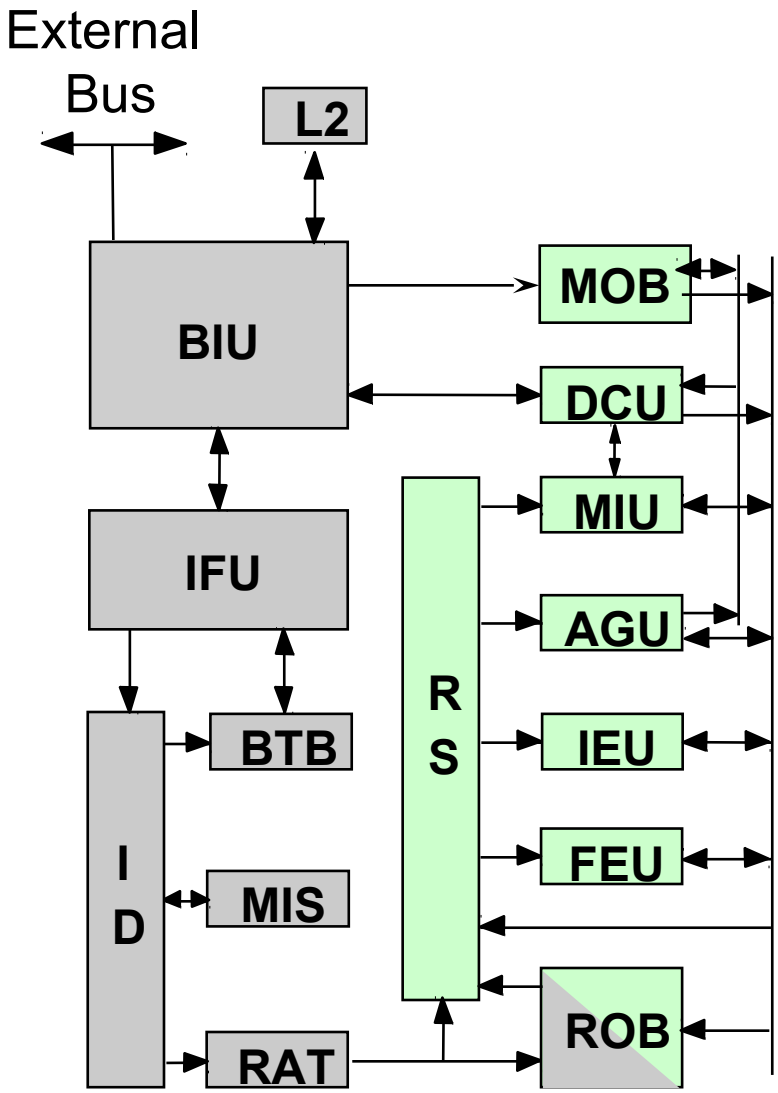
Micro Operations (Uops)

- **Each CISC inst is broken into one or more RISC *uops***
  - Simplicity
    - Each uop is (relatively) simple
    - Canonical representation of src/dest (2 src, 1 dest)
  - But increased instruction count
- **Simple instructions translate to a few uops**
  - 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

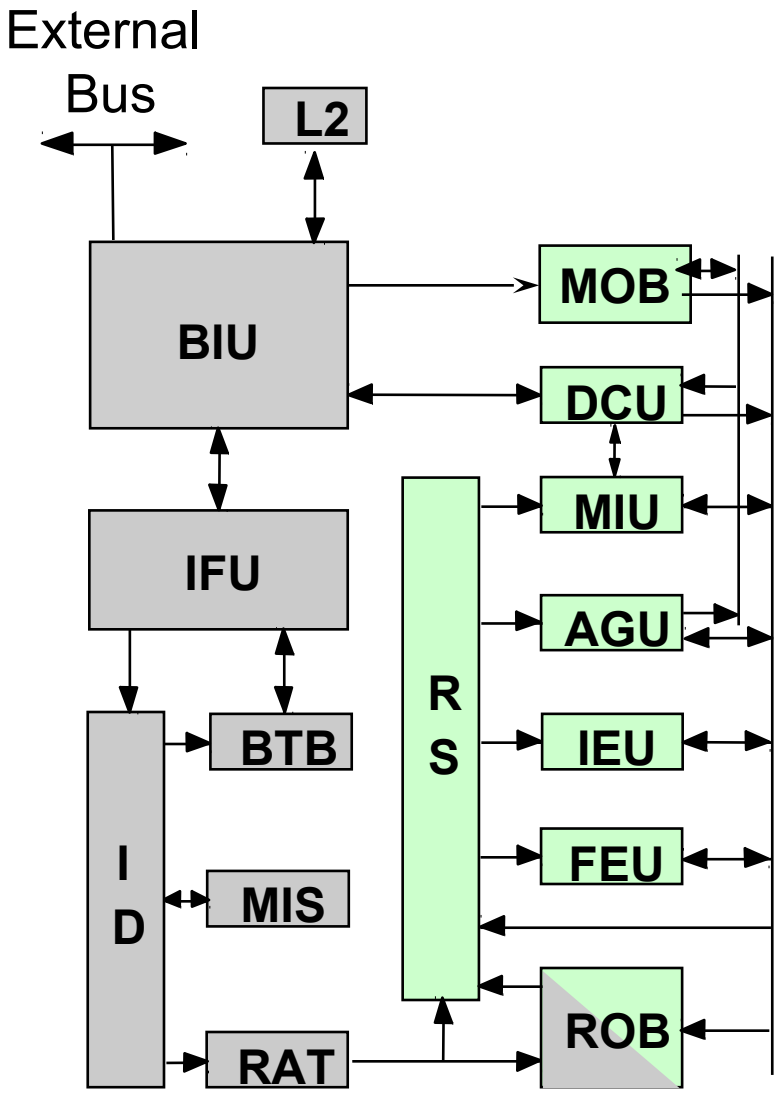
- **Complex instructions need ucode**

Out-of-order Core: ROB + RS



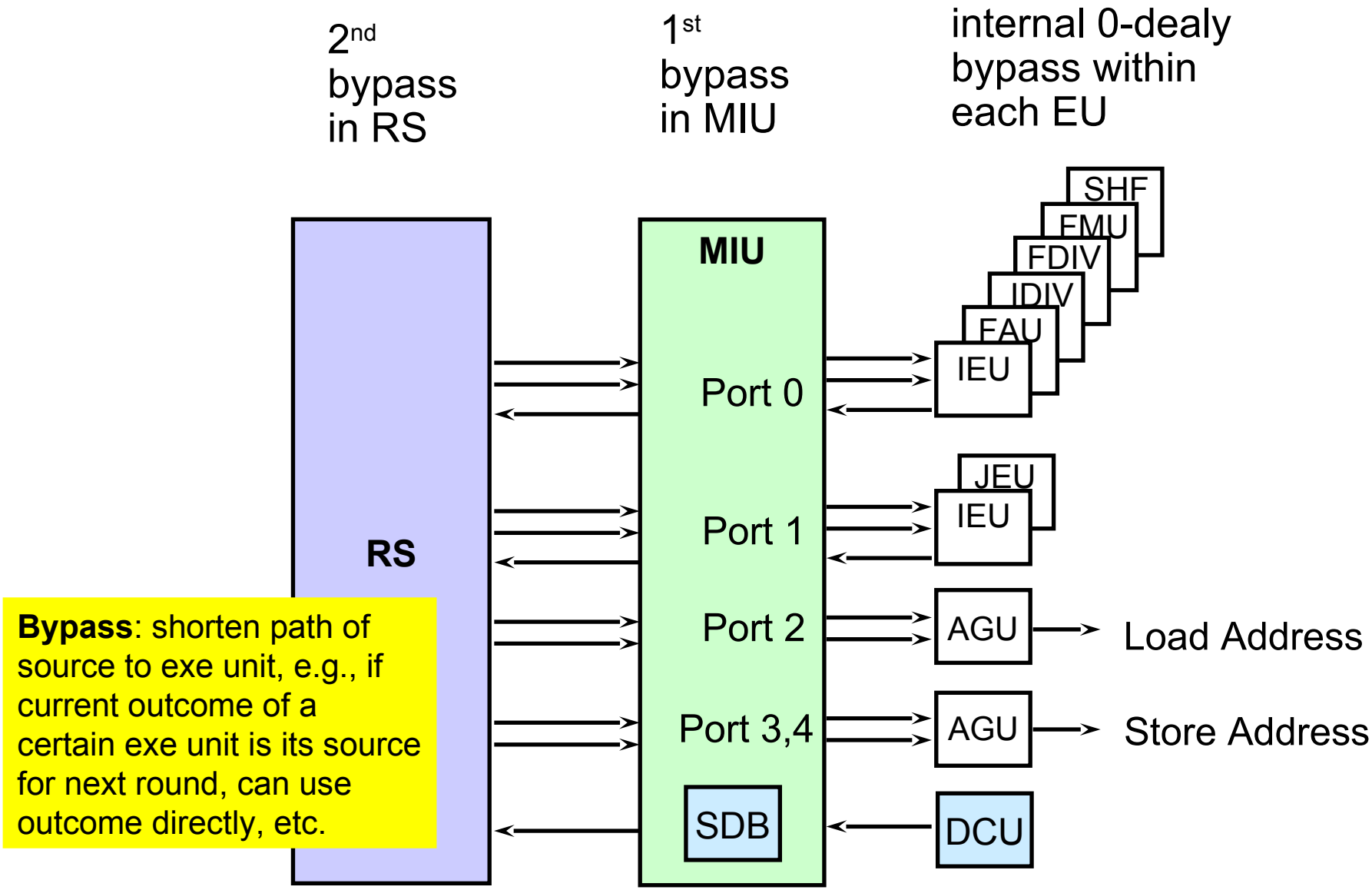
- **Reorder Buffer (ROB):**
  - Holds “not yet retired” instructions
  - 40 ordered entries (cyclic array)
  - Retired in-order
  - It’s possible some instruction already executed (their result known), but cannot be retired since
    - still have speculative status
    - and/or are waiting for previous instructions to retire in order
- **Reservation Stations (RS)**:
  - Holds “not yet executed” instructions
  - 20 entries (subset of ROB)
  - Up to 4 simultaneous ops can get in and out of RS simultaneously
- **After execution**
  - Results written to both ROB & possibly to RS (when source of other instructions)

Out-of-order Core: execution units



- **Basic execution Units**
  - IEU: Integer Execution Unit
  - FEU: Floating-point Execution Unit
- **Memory-related execution units**
  - DCU: Data Cache Unit
  - AGU: Address Generation Unit
    - Computes effective address & sends to DCU to fetch from cache)
  - MOB: Orders memory loads and stores
  - L2: Level 2 cache
  - MIU: Memory Interface Unit
- **MIU feeds execution units**
  - Positioned between RS & exe units
  - Has 5 ports (see next slide)
  - Some units may be similar between ports, others exclusive

Out-of-order Core: execution units



Bypass: shorten path of source to exe unit, e.g., if current outcome of a certain exe unit is its source for next round, can use outcome directly, etc.

RAT & ALLOC

- **There are ≤ 4 new uops/cyc; for each such uop**
  - Perform register allocation & renaming, specifically…
- **For each new uop , use RAT (Register Alias Table) to**
  - Source reg(s): map arch reg(s) to physical reg(s)
    - arch reg => latest phys reg that updated arch reg
  - Target reg: (**1**) allocate new phys reg; (**2**) update RAT accordingly
    - Now arch reg points to newly allocated phys reg (for next time)

RAT:

arch reg	phys reg#	location
EAX	0	RRF
EBX	19	ROB
ECX	23	ROB

- **The Allocator (Alloc)**
  - Assigns each uop with new ROB & RS entries
  - Write up the matching phys regs to RS (along with the rest of the uop)
  - Allocate Load & Store buffers in the MOB (for load & store ops)

Reorder Buffer (ROB)

- **Hold 40 uops which are "not yet committed"**
  - Same order as program (cyclic array)
  - Provide large physical register space for reg renaming
  - A physical register is actually an item within a matching ROB entry
    - phys reg number = ROB entry number
    - phys reg = uop's target destination (there's always exactly one)
    - phys regs buffer the execution results until retirement

#entry	entryValid	dataValid	data (physical reg)	arch target reg
0	1	1	12H	EBX
1	1	1	33H	ECX
2	1	0	xxx	ESI
…				
39	0	0	xxx	XXX

  - *Valid data* is set after uop executed (& result written to physical reg)

RRF – Real Register File

- **Holds the Architectural Register File**
  - Architectural Register are numbered: 0 – EAX, 1 – EBX, …
  - This is "the state" of the chip (can't roll back)
- **The value of an architectural register**
  - Is the value written to it by the last **committed** uop (which writes to that reg)
  - So long as we don't change the RRF, we don't change the state

RRF:

#entry	Arch Reg Data
0 (EAX)	9AH
1 (EBX)	F34H

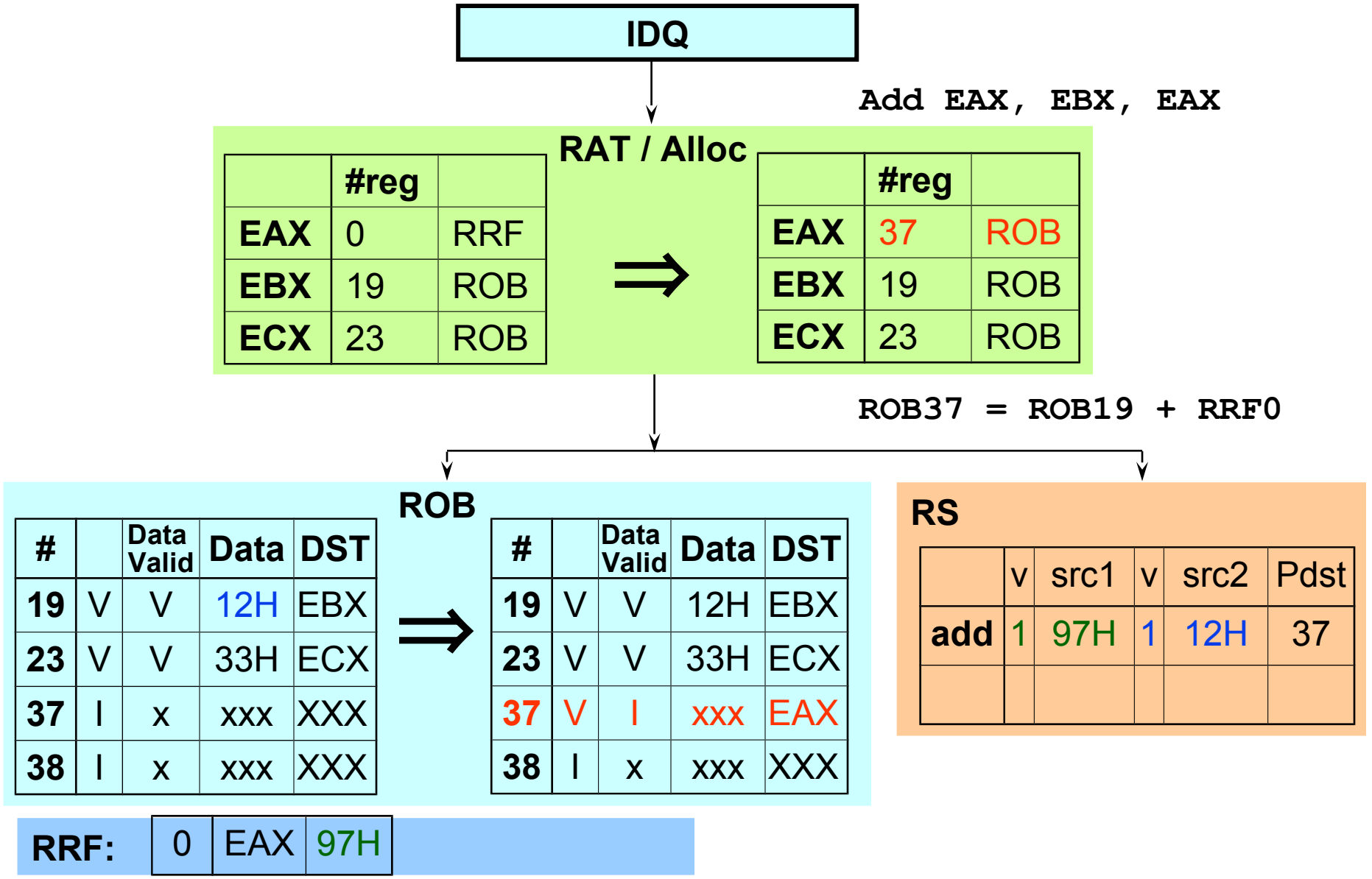
Uop flow through the ROB

- **Uops are entered in order**
  - Registers renamed by the entry # (there's a head and a tail)
- **Once assigned**
  - Execution order unimportant
- **After execution:**
  - Entries marked *"executed" (dataValid=1)* & wait for retirement
  - Retirement occurs once all prior instruction have retired
  - => Commit architectural state only after speculation was resolved
- **Retirement**
  - Detect exceptions and misprediction
    - Branch result might impact uops down the road
    - Initiate repair to get machine back on track
  - Update *"real" regs* (in RRF) with value of renamed (phys) regs
  - Update memory
  - Clear ROB entry

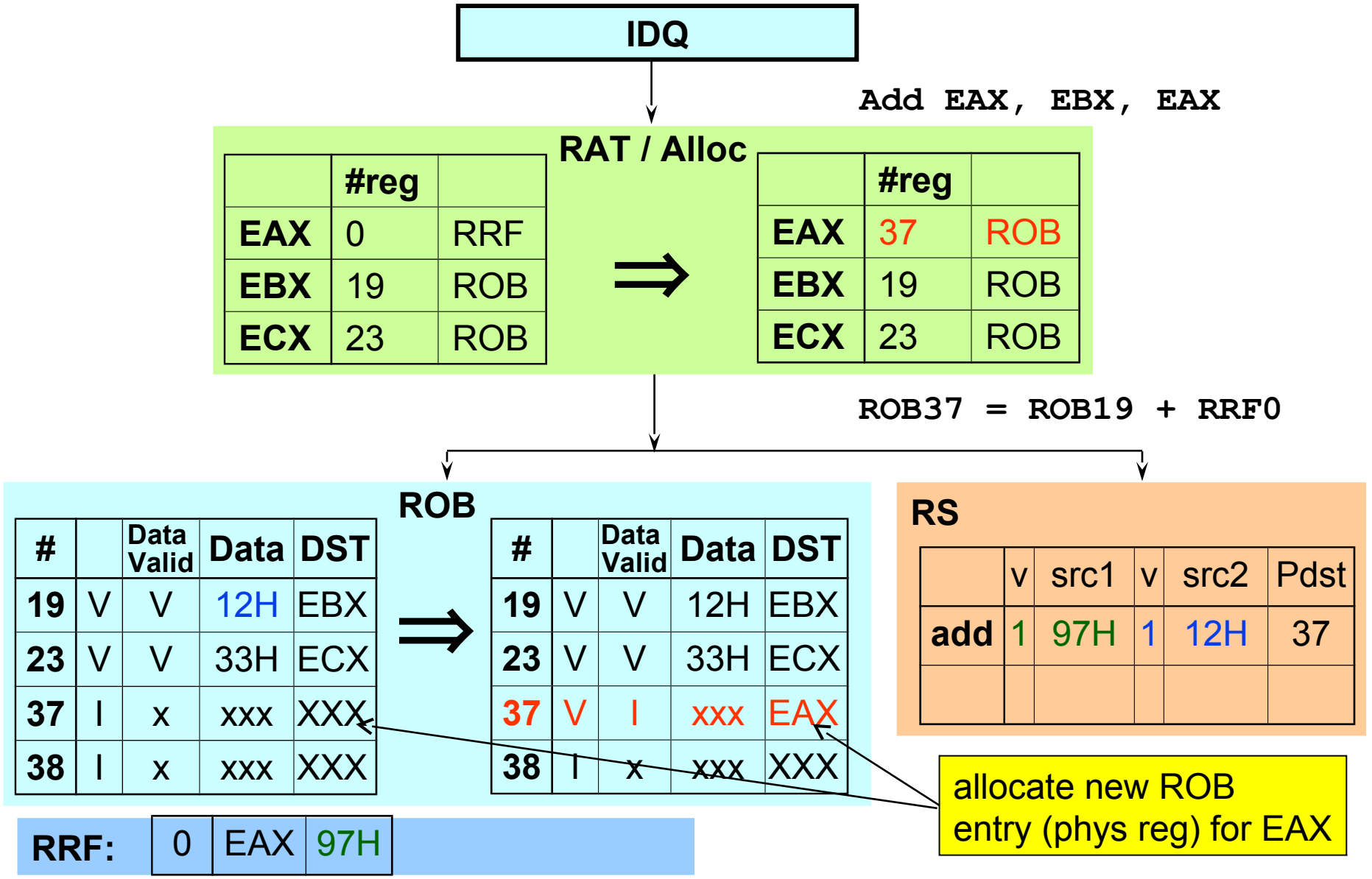
Reservation station (RS)

- **Pool of all "not yet executed" uops**
  - Holds the uop code & source data (until it is dispatched=scheduled)
- **When a uop is allocated in RS, operand values are updated**
  - If operand is arch reg => value taken from the RRF
  - If operand is phys reg (with *dataValid =1*) => value taken from ROB
  - If operand is phys reg (with *dataValid=0)* => wait for value
- **The RS maintains operands status "*ready* / *not-ready*"**
  - Each cycle, executed uops make more operands *"ready"*
    - RS arbitrates WB busses between exe units
    - RS monitors WB bus to capture data needed by waiting uops
    - Data can bypass directly from WB bus to exe unit (like we've seen)
  - Uops whose all operands are *ready* can be *dispatched*
    - Dispatcher chooses which *ready* uop to execute next
    - Dispatcher sends chosen uops to appropriate functional units
    - Need appropriate functional unit to be vacant

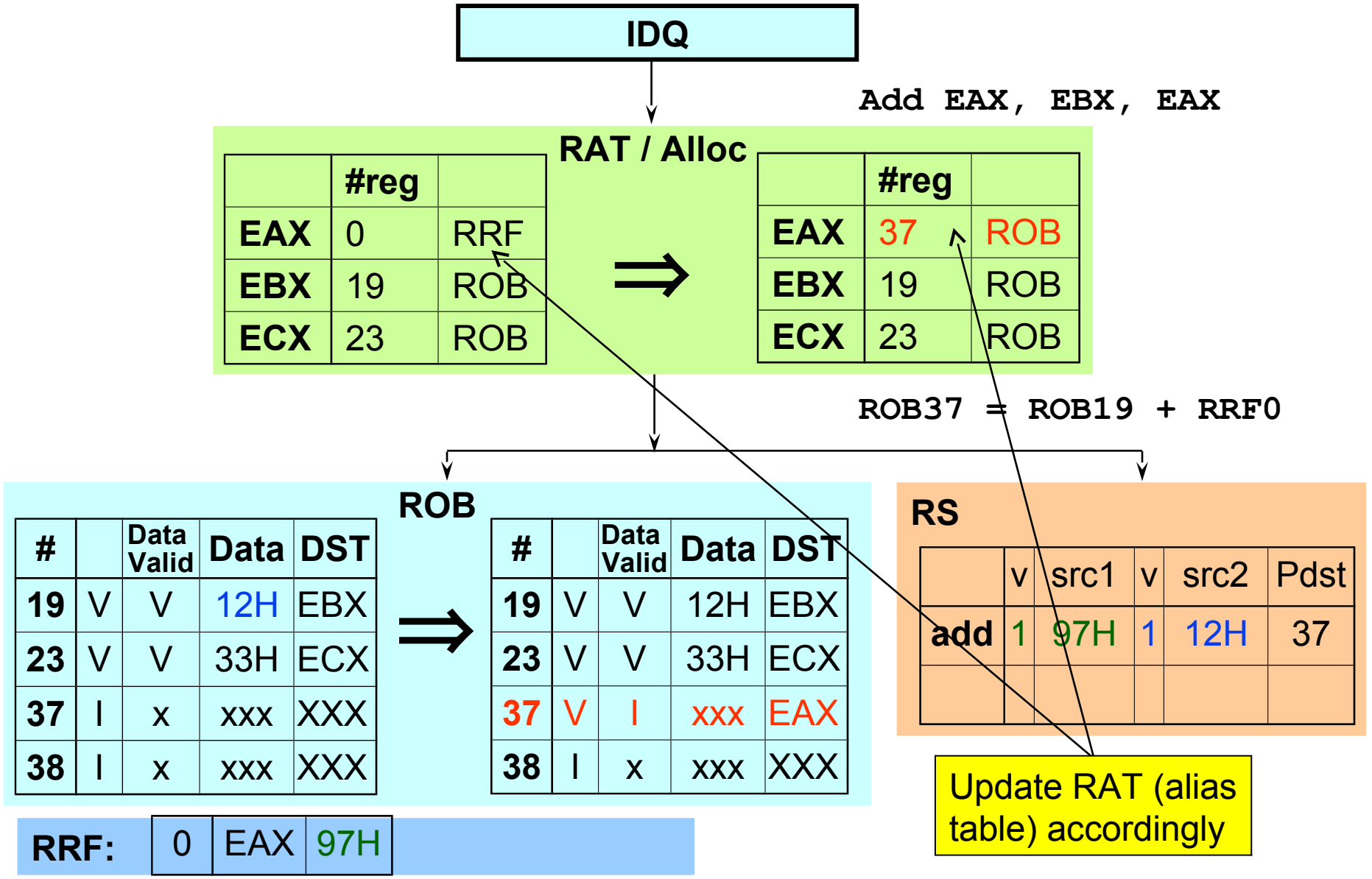
Register Renaming example



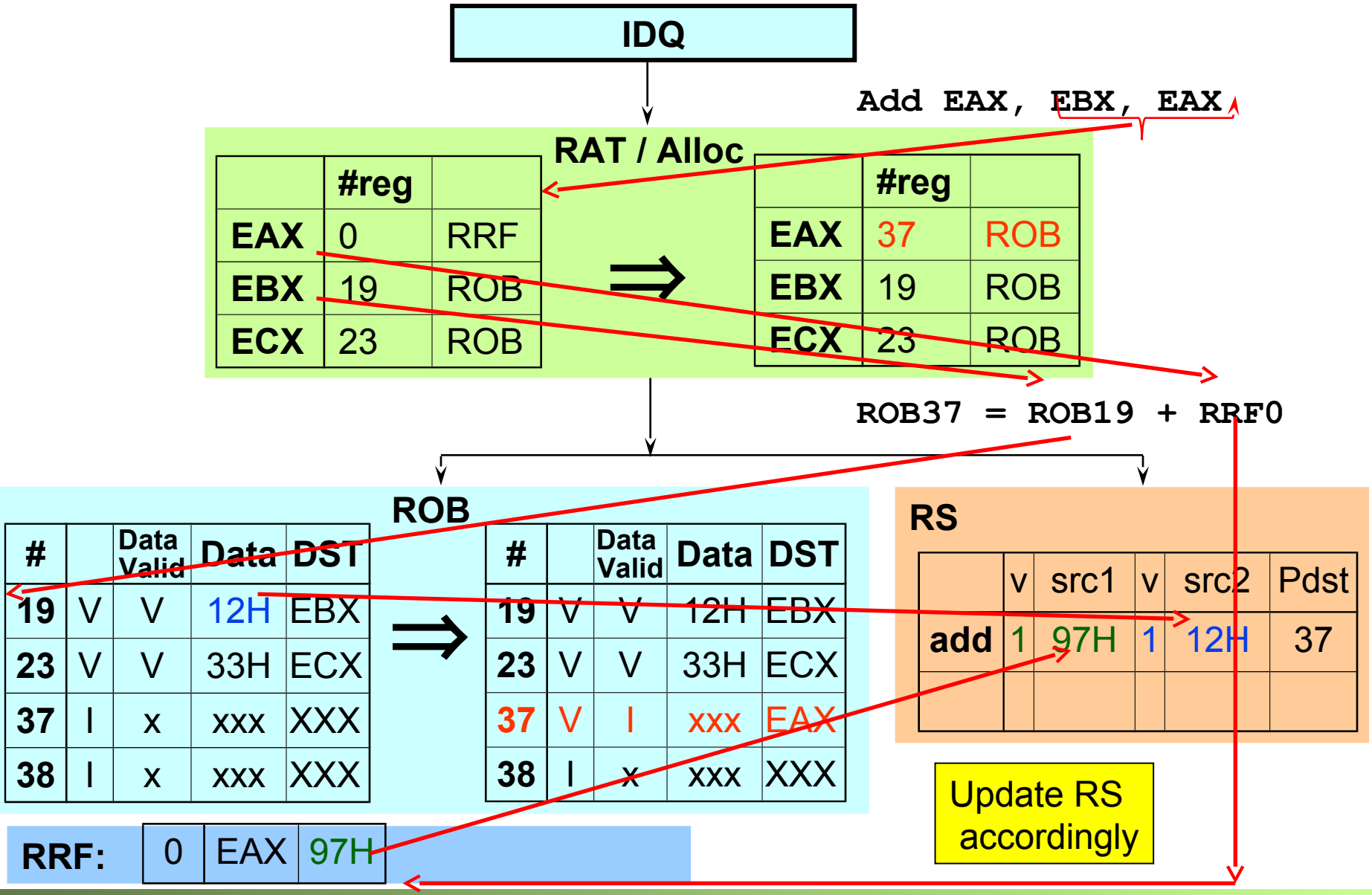
Register Renaming example



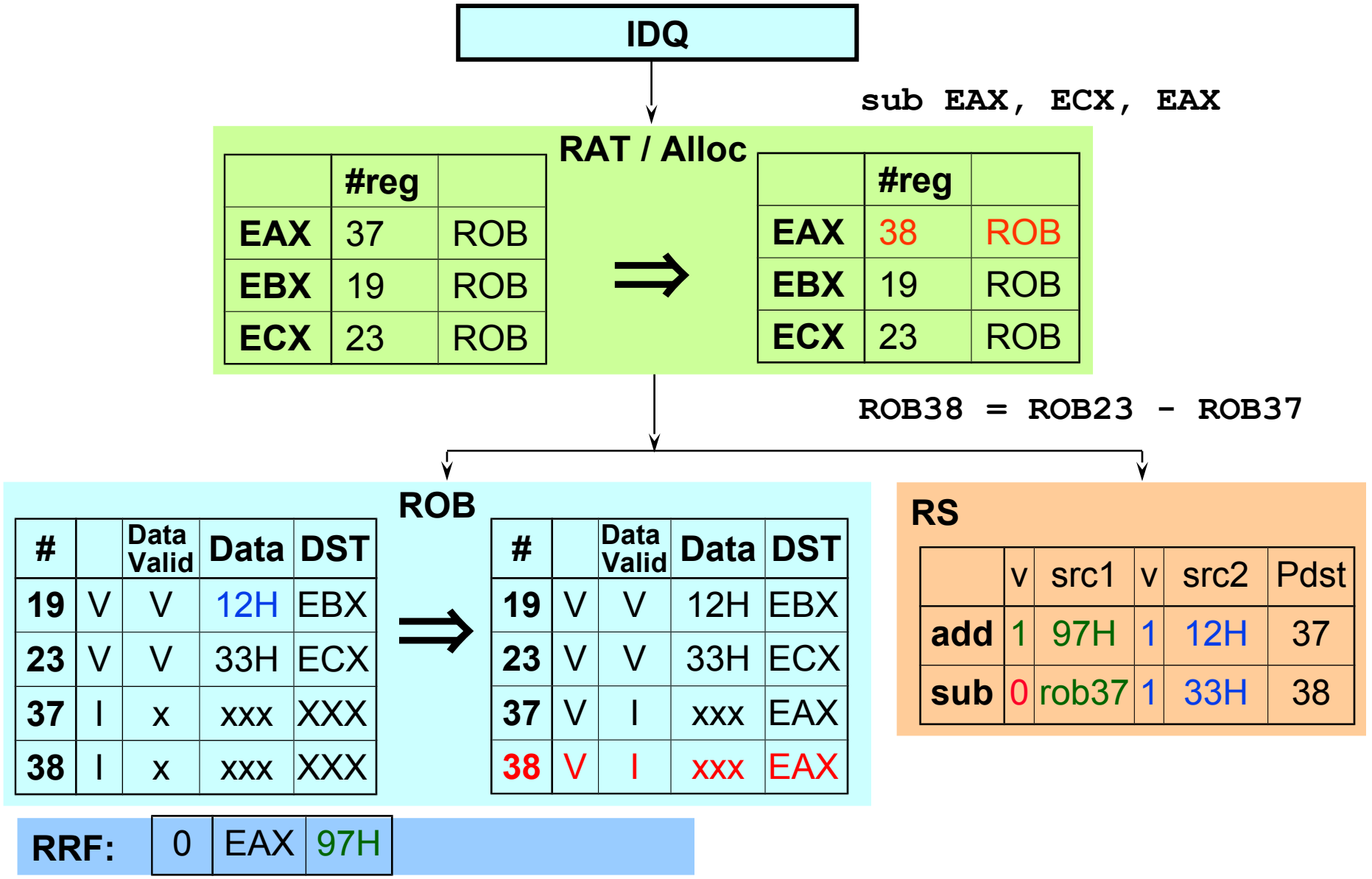
Register Renaming example



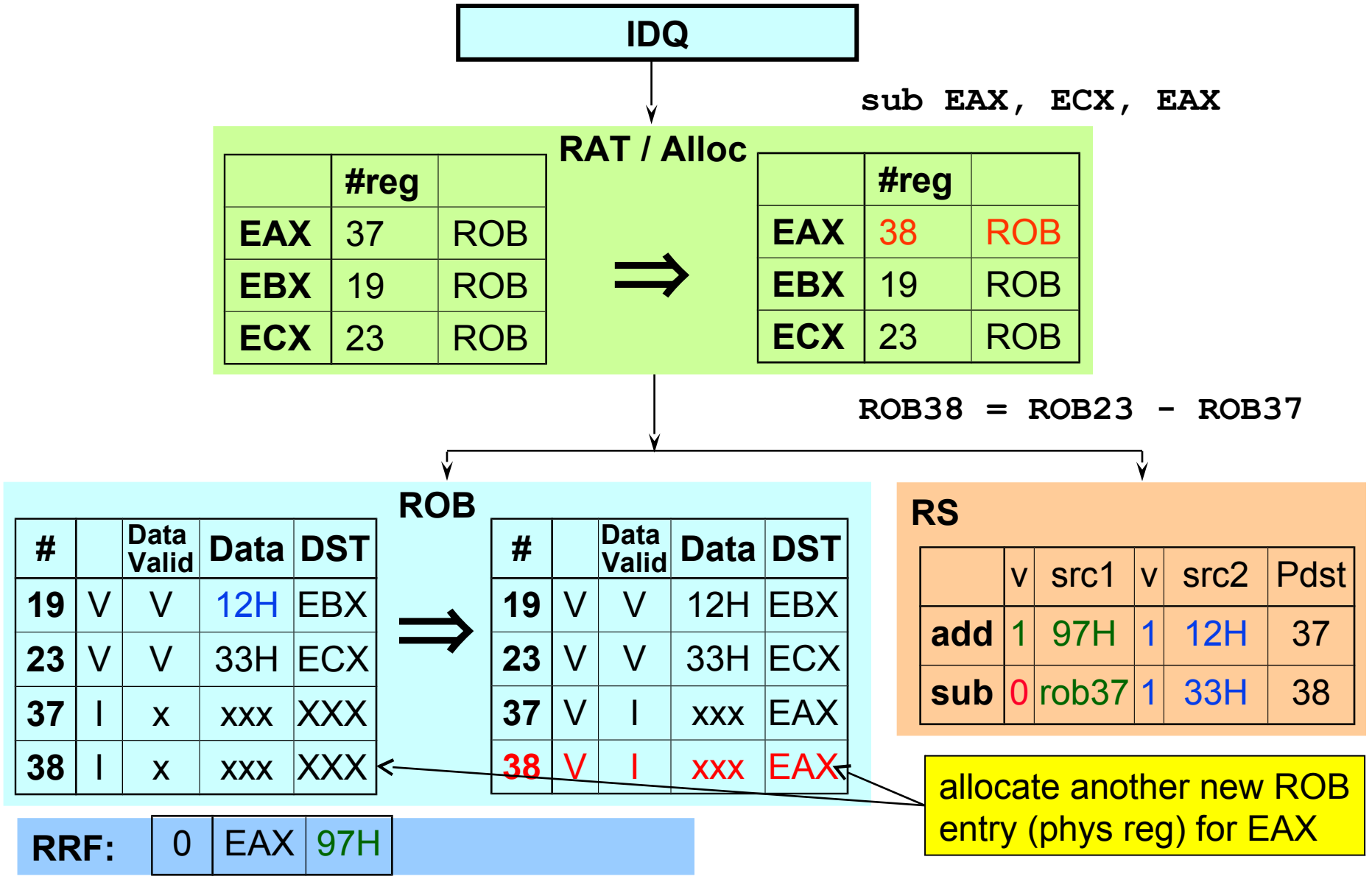
Register Renaming example



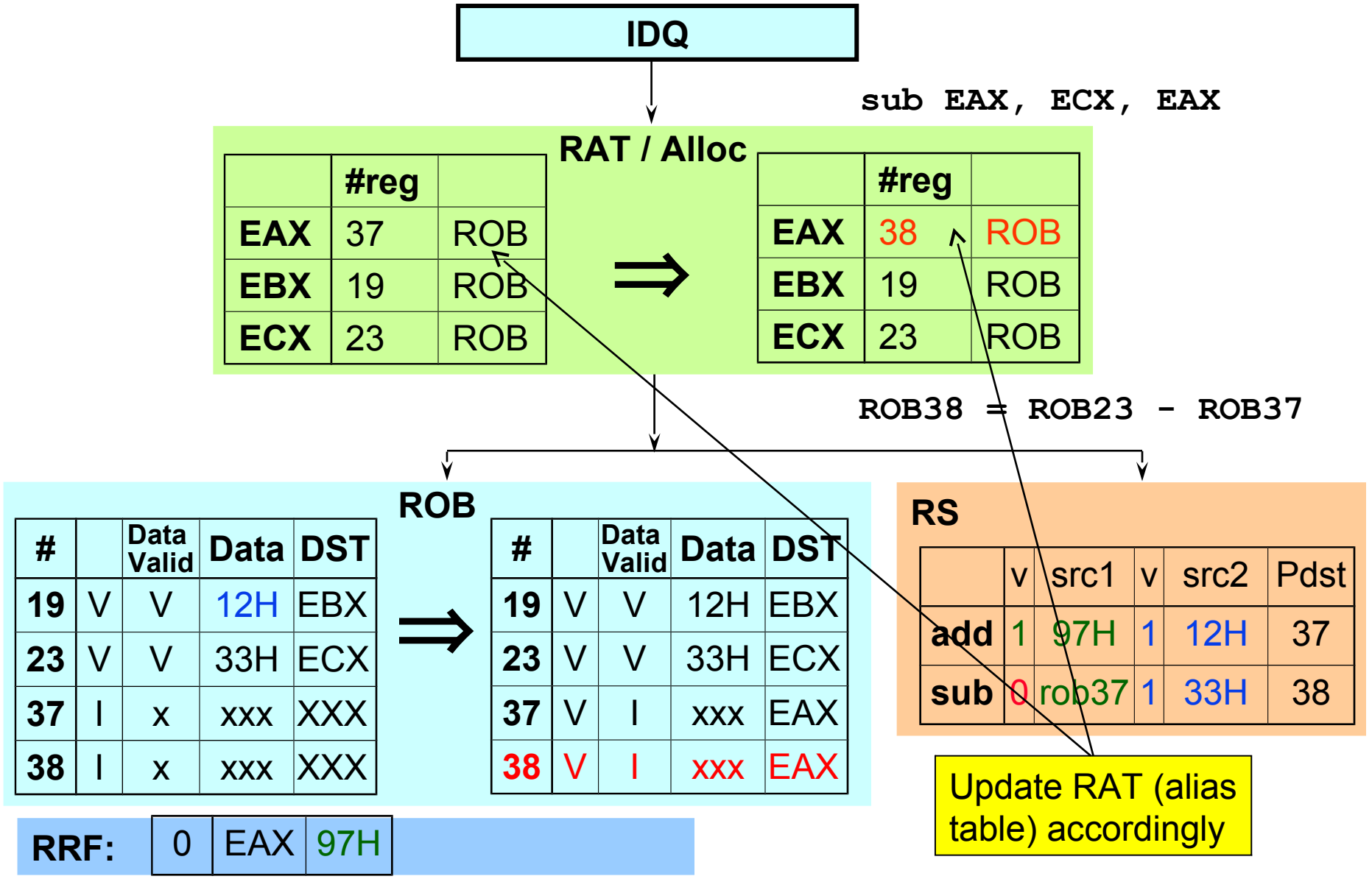
Register Renaming example (2)



Register Renaming example (2)



Register Renaming example (2)



Register Renaming example (2)