

# **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 20, 2004**

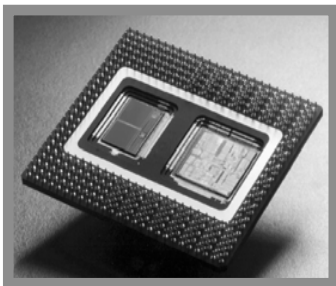
# Agenda

- **General Information**
- **μarchitecture basics**
- **Pentium<sup>®</sup> Pro Processor μarchitecture**
- **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.)

- Coppermine (Pentium-III - 2000)
- Core @1000MHz
- 256KB L2 on chip @ 1000MHz
- Performance:
  - ▮ >46 SPECint95
  - ▮ >20 SPECfp95
- 0.18  $\mu\text{m}$  CMOS process
- ~20M transistors
- Tualatin (Pentium-III - 2002)
- Core @1400MHz
- 512KB L2 on chip @ 1400MHz
- Performance (estimated):
  - ▮ >60 SPECint95
  - ▮ >30 SPECfp95
- 0.13  $\mu\text{m}$  CMOS process
- ~44M transistors

- Pentium M Processor Banias 2003
- Core @1800MHz
- 1024KB L2 on chip @ 1800MHz
- Performance (estimated):
  - ▮ >80 SPECint95
  - ▮ >50 SPECfp95
- 0.10  $\mu\text{m}$  CMOS process
- ~77M transistors

# 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

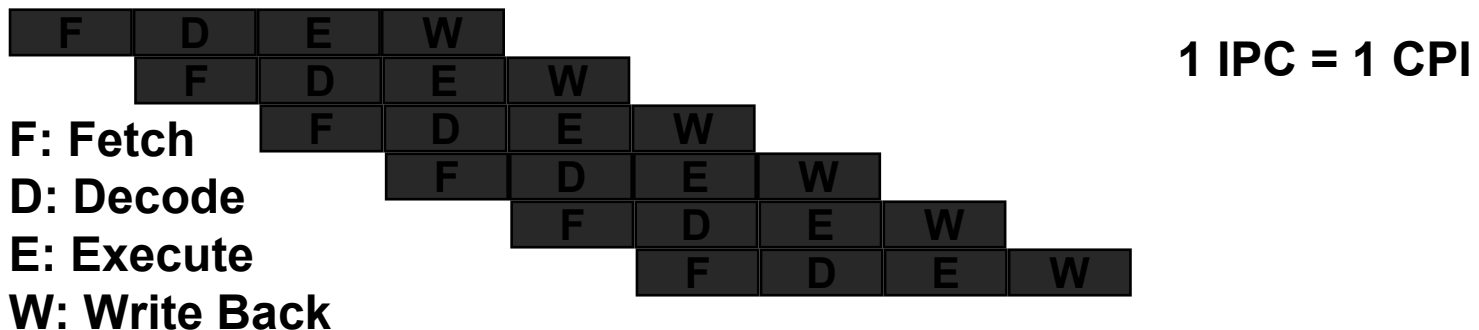
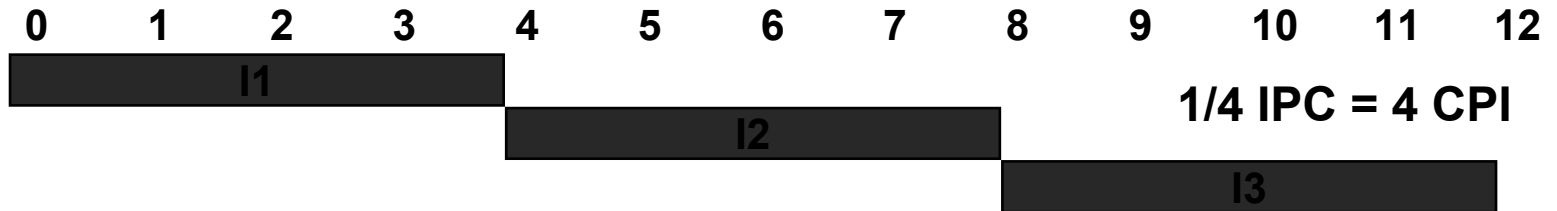
**Skip to  $\mu$ arch**

# 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

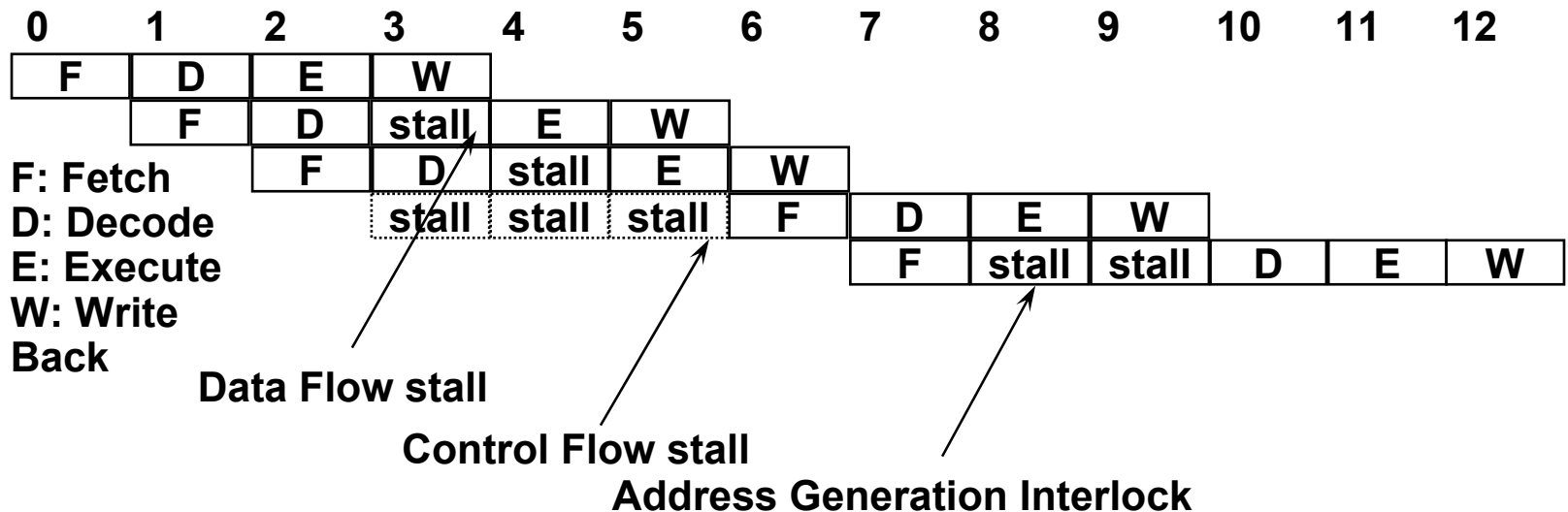


- Increased throughput

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

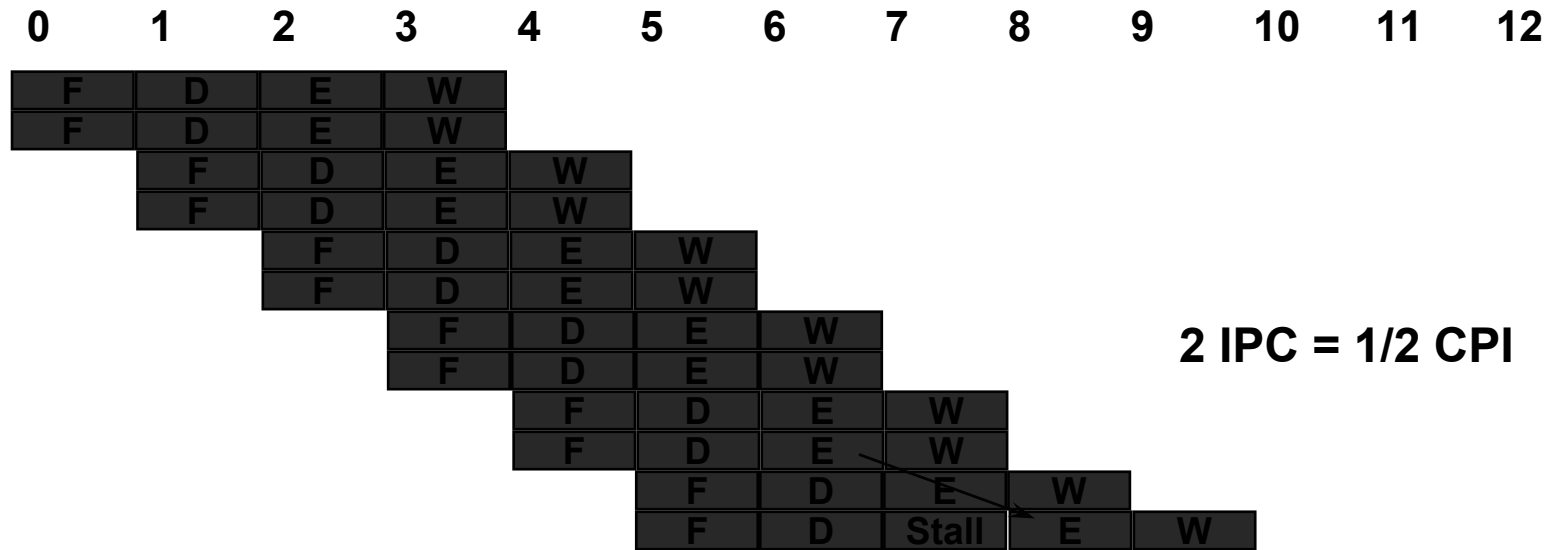
# Pipeline Stalls

- 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

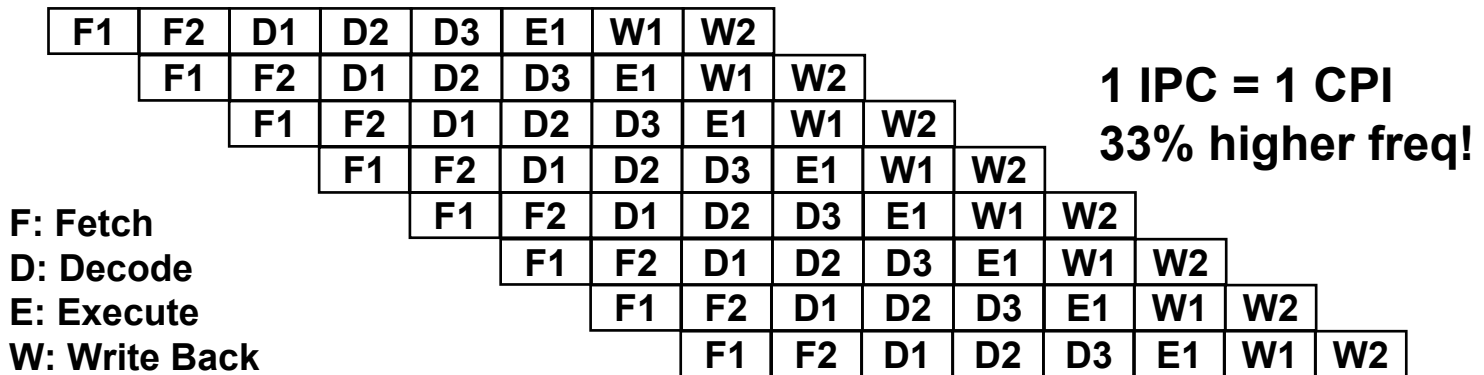


- Ideally, can multiply the throughput
  - but stall penalty is increased

# 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%





# 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

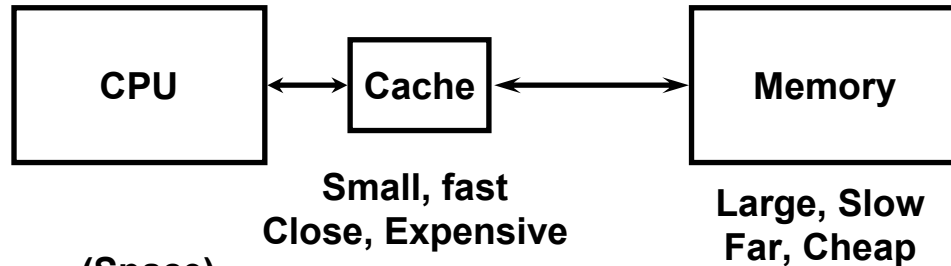
- Solution: Cache - A Small, Fast, Close memory
  - Serves as a buffer between CPU and main memory

- Contains copy of a portion of the main memory

- Small in size
- Dynamically changed

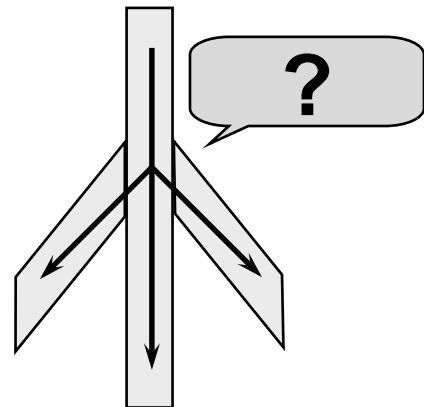
- Exploit space and time locality:

- 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

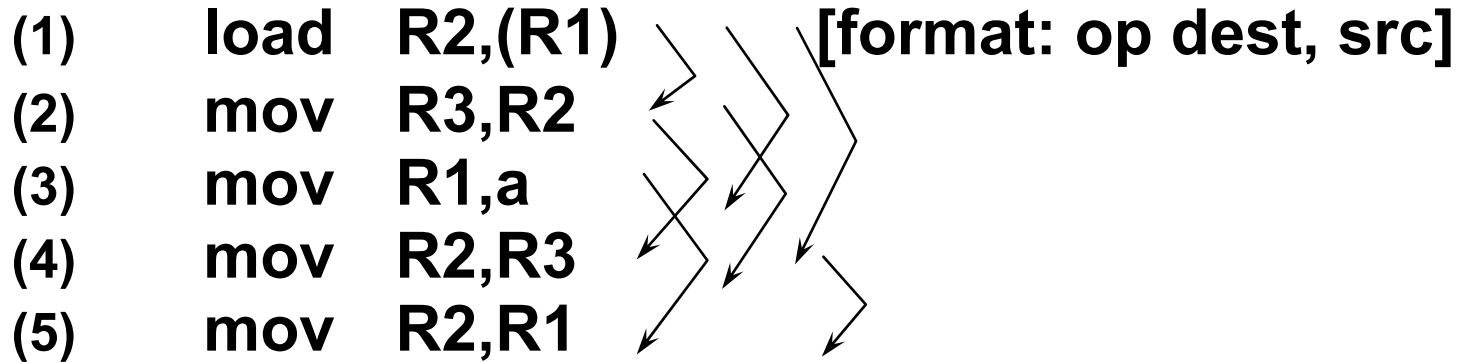
# Speculative Execution

- **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)	add R2,R1	add	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?**

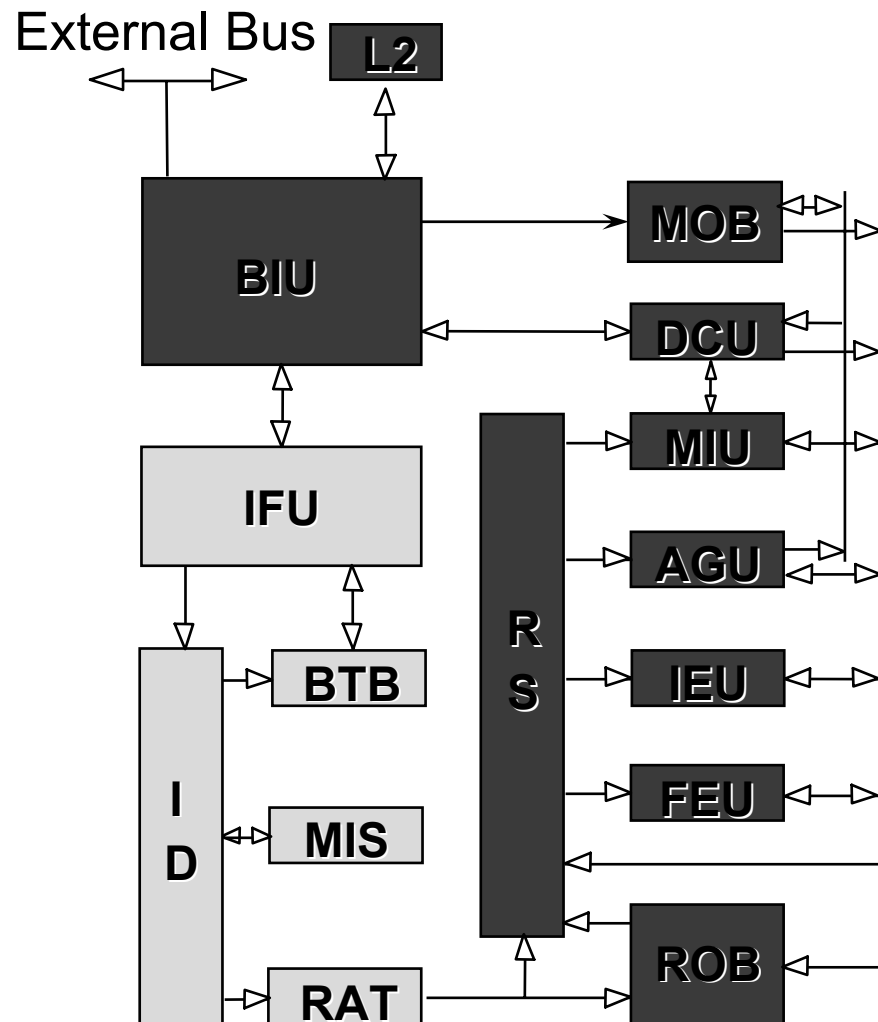
# Micro Operations (Uops)

- 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



# Microarchitecture



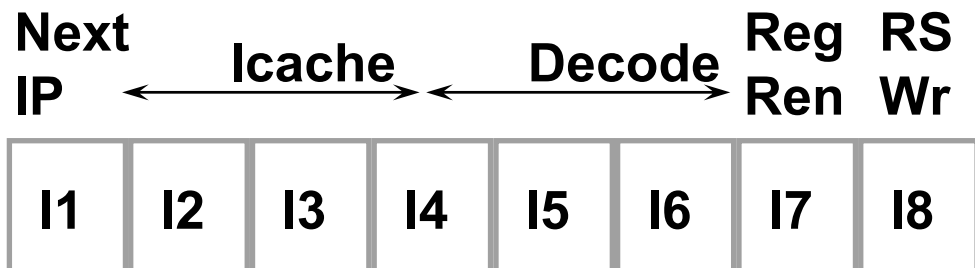
## ● In-Order Front End

- 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

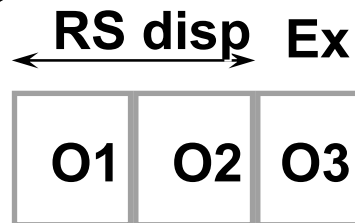
- Implementation
  - Use local history to predict direction
  - Need to predict multiple branches
  - ⇒ Need to predict branches before previous branches are resolved
  - ⇒ Branch history updated first based on prediction, later based on actual execution (speculative history).
  - Target address taken 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!

# Pipeline

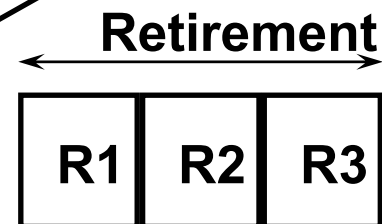


- In-Order Front End

- 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