Chapter 3: SuperScalar Concepts

Prof. Ben Lee
School of Electrical Engineering and Computer Science
Oregon State University

Chapter Outline

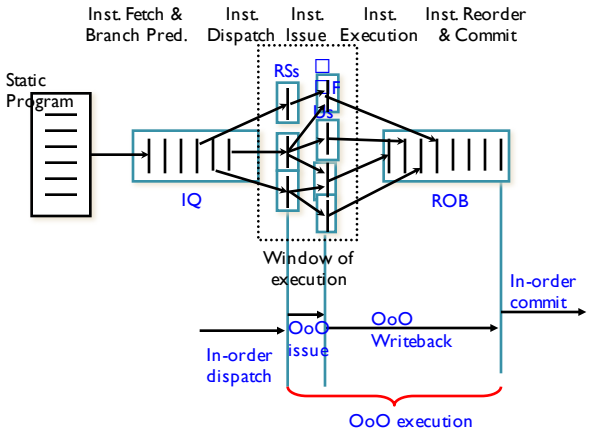
- Elements of a SuperScalar Processor
- SuperScalar Execution Model
- Complexity of SuperScalar Processors
- Case Study - Pentium 4
- Microprocessor Comparison
- The Core Microarchitecture

Elements of a SuperScalar Processor

Elements of a SuperScalar Processor

- All the elements listed below are integrated in a cohesive, almost seamless, manner:
  - Fetching multiple instructions and by predicting the outcomes of, and fetching beyond, conditional branches.
  - Determining RAW dependences among register values.
  - Issuing multiple instructions OoO and resolving WAR and WAW hazards.
  - Multiple FUs and memory hierarchy capable of simultaneous servicing memory references.
  - Committing the process state in correct order (precise interrupt).

Basic SuperScalar Design (1)



Basic SuperScalar Design (2)

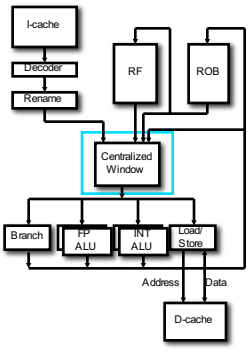
- Instruction Window – Consists of Instruction Queue (IQ), and Reservation Stations (RS), which allows multiple instructions to be executed out-of-order, i.e., OoO issue and OoO writeback, on multiple FUs.
- Register Renaming - Using additional operand buffers in RS and/or ROB eliminates WAR and WAW hazards.
- Reorder Buffer (ROB) - Allows the process state to be committed in correct order by dispatching in-order and committing in-order.

Instruction Window

- Instruction Window (IW) resides between the instruction decoder and the FUs. The decoder places instructions into the window, and the instruction-issue logic examines instructions in the window to appropriately select instructions to issue to the FUs.
- There are three ways to implement the instruction window:
  - Centralized
  - Distributed based on type
  - Distributed based on FUs

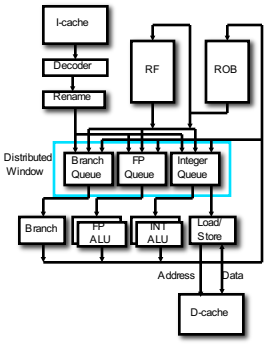
Different IW Implementations

Centralized



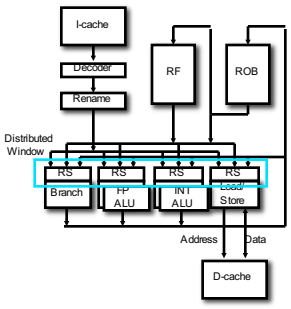
Examples: UltraSparc, Pentium Pro (II & III), Core

Distributed based on type



Examples: Alpha, MIPS, Pentium 4, Power5

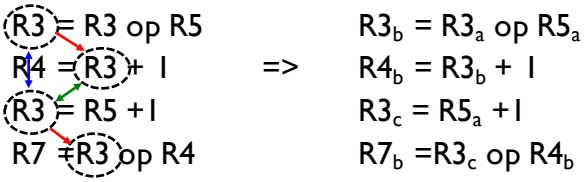
Distributed based on FUs



Examples: PowerPC

Register Renaming

- The processor allocates a new register for every new value produced, i.e., for every instruction that writes a register.



- Need more physical registers than logical registers.
- Renaming avoids WAW and WAR hazards; turns instruction stream into a "single assignment" form.

Register Renaming in Tomasulos' Approach (1)

- In the Tomasulo Algorithm, we saw that renaming was implemented by
  - Operand buffers in the RSs that serves as extra registers.
    - This eliminates the need to get the operand from the RF (which caused WAR and WAW in the first place).
  - A pending instruction designates the RS that will provide its input.
  - As instructions are issued, the register specifiers for pending operands are renamed to the names of the RS.
  - When successive writes to a register appear, only the last one is actually used to update the register.

Register Renaming in Tomasulos' Approach (2)

Example: WAR hazard

Instruction		Issued at	Execute During	Write Result	Comments
L.D	F6,34(R2)	0	1-2	3	Load unit is busy until cycle 3
L.D	F2,45(R3)	1	4-5	6	Load unit is busy until cycle 6
MUL.D	F0,F2,F4	2	7-16	17	wait for F2; MULT1 busy until cycle 17
SUB.D	F8,F6,F2	3	7-8	9	wait for F2; ADD1 busy until 9
DIV.D	F10,F0,F6	4	18-57	58	wait for F0 of MULTF; MULT2 busy until 58
ADD.D	F6,F8,F2	5	10-11	12	wait for F8 of SUBF; ADD1 busy until 12

- Both `DIV.D` and `ADD.D` were issued, even though there is a WAR hazard on `F6`.
  - If the `L.D` has completed, then V_k (source operand `F6`) will store the result.
  - If the `L.D` had not completed, then Q_k (RS that will produce `F6`) would point to the Load1 reservation station.

Register Renaming in Tomasulos' Approach (3)

Example: WAW hazard

```
Loop: LF     F0, 0(R1)
      MULTF  F4, F0, F2
      SF     0(R1), F4
      SUBI   R1, R1, #8
      BNEZ   R1, Loop
```

```
LF     F0, 0(R1)
MULTF  F4, F0, F2
SF     0(R1), F4
SUBI   R1, R1, #8
BNEZ   R1, Loop
LF     F0, 0(R1)
MULTF  F4, F0, F2
SF     0(R1), F4
SUBI   R1, R1, #8
BNEZ   R1, Loop
...
```

WAW

- Will be in different RSs and the RS that will produce results will be different.
- This is done using Q_i in the RF that indicates which RS will produce the result.

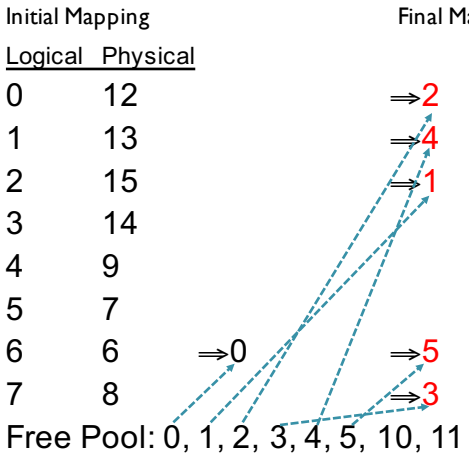
Register Renaming Using Physical Registers (1)

- Since not all processors use RSs, another way to do it is by have physical registers > logical registers.
- Example:

```
L.D   F6, 34(R2)
L.D   F2, 45(R3)
MUL.D F0, F2, F4
SUB.D F7, F6, F2
DIV.D F1, F0, F6
ADD.D F6, F7, F2
```

Assume 8 logical, 16 physical registers.

Register Renaming Using Physical Registers (2)



Register Renaming Using Physical Registers (3)

- Code with physical register assignments

```
L.D   P0, 34(R2)
L.D   P1, 45(R3)
MUL.D P2, P1, P9
SUB.D P3, P0, P1
DIV.D P4, P2, P0
ADD.D P5, P3, P1
```

- Afterwards, reclaim the physical registers that are no longer used.
  - Ignoring precise interrupts, a register can be returned after the last read is done.
  - Considering precise interrupts, a register can be returned after it is logically over-written.

Register Renaming Using ROB (1)

- The ROB is managed as a FIFO queue:
  - When an instruction is decoded, it is allocated an entry at the tail of the ROB so that result value of this instruction can be written into the allocated entry after it completes.
  - When the value reaches the head of the ROB, it is written into the RF (if no exception occurred). If the instruction is not complete when its entry reaches the bottom of the ROB, the ROB does not advance until the instruction completes.
- ROB keeps the original lexigraphical order of the instructions and allowed out-of-order execution while preserving in-order writing of results to registers and memory => precise interrupts!

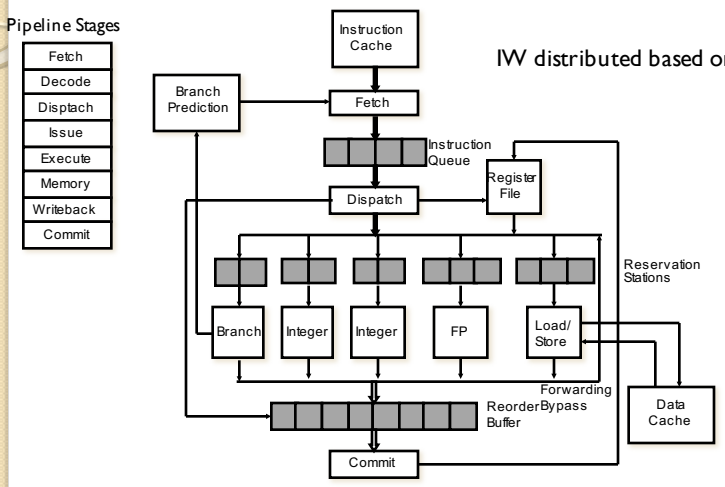
Register Renaming Using ROB (2)

- Renaming with ROB:
  - Logical registers == physical registers
  - ROB commits state in order
  - ROB holds "renamed" logical registers
  - Registers read only at dispatch
  - Otherwise, values come from FUs or ROB
- Reorder buffer while FP divide in progress:

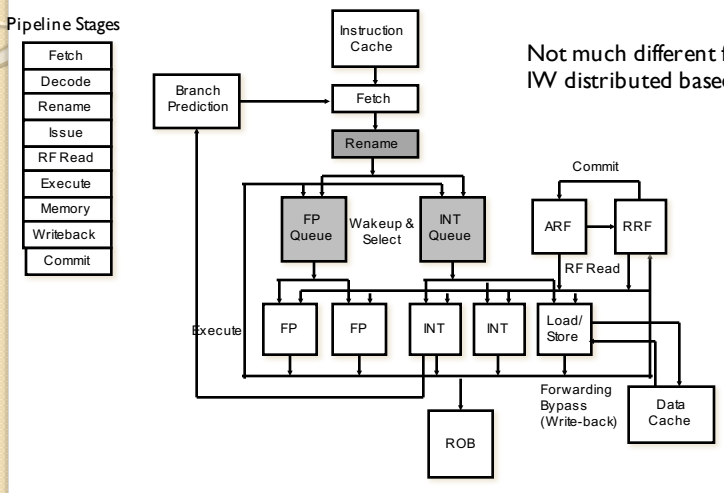
		Entry	Content
L.D	F6, 34(R2)	0	F6
L.D	F2, 45(R3)	1	F2
MUL.D	F0, F2, F4	2	F0
SUB.D	F7, F6, F2	3	F7
DIV.D	F1, F0, F6	4	F1
ADD.D	F6, F7, F2	5	F6

SuperScalar Execution Model

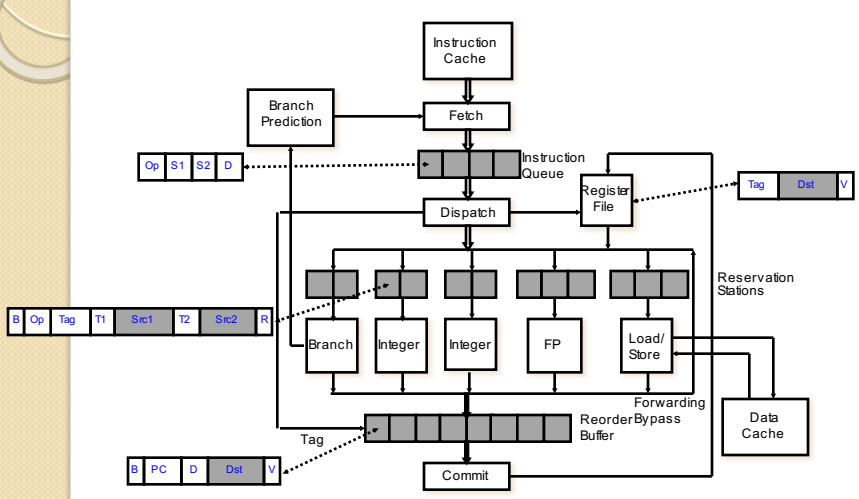
SuperScalar Execution Model



SuperScalar Execution Model



Information in IQ, RS, & ROB



Information in Various Components

Instruction Queue:

Op Opcode

S1/2 Source 1/2 register number.

D Destination register number.

Reservation Station:

B Indicates the RS has been allocated.

Op Opcode

Tag The pointer to the ROB entry where the result of this instruction will be written.

Src1/2 Source operand.

T1/2 The pointer to the ROB entry which will receive the result needed by the source registers Src1/2.

R Indicates the instruction is ready to be issued.

FU:

B Indicates FU is busy.

Op Opcode

Tag The pointer to the ROB entry where the result of this instruction will be written to.

Src1/2 Source operand.

Reorder Buffer:

B Indicates the ROB entry is busy.

PC Address of the instruction

D Destination register number.

Dst Result value

V When true, signifies that Dst field holds a valid result.

Register File:

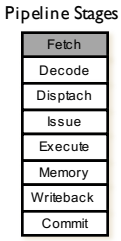
Tag The pointer to the ROB entry that will contain the result.

Dst Result value.

V When true signifies that Dst field holds a valid result.

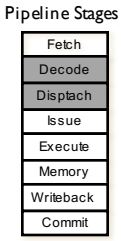
Fetch Stage

- Instructions are fetched from the I-cache and enqueued into the IQ in program order.
- For an *n*-issue superscalar operation, the Fetch stage needs to fetch at least *n*-instructions.



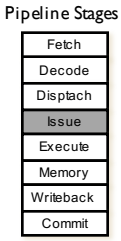
Decode/Dispatch Stage

- Instructions are decoded and then dispatched when RSs are available:
  - Up to *n* instructions can be decoded and dispatched per cycle.
- For each instruction dispatched
  - Allocate a RS and set it to busy.
  - Allocate the next sequential entry in the ROB and set it to busy. Place the Tag value pointing to this entry into the Tag field of the allocated RS.
  - For destination register D, update its Tag with the Tag of the just allocated ROB entry and set Dst to invalid.
  - For each of the source registers S1 and S2, check its Dst in RF.
    - If valid, read Dst from register and update Src1/Src2.
    - Else, check if Dst of ROB entry pointed to by its Tag is valid:
      - If valid, update Src1/Src2 with Dst;
      - Else, update T1/T2 with Tag from the register.



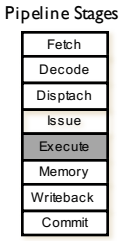
Issue Stage

- Issue all the instructions when both source operands and FUs are available (i.e., R-bit is set). This is done by
  1. Copy Op, Tag, Src1, and Src2 from RS to FU. Set FU to busy.
  2. Free RS.



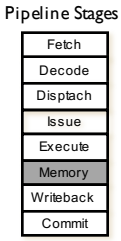
Execute Stage

- Execute instructions.
  - Latency will depend on the instruction



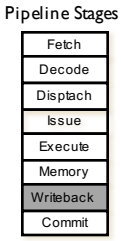
Memory Stage

- Only load and store instructions perform this stage.



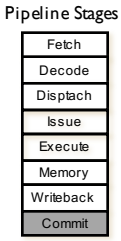
Write-back Stage

- When an instruction completes, post its result in the ROB and forward.
  - Write the result into the Dst field of the entry in the ROB pointed to by the Tag field, and set it to valid.
  - Forward the result back to RSs where the Tag field of the completed instruction matches source register tags T1 and T2 in RSs. For each match, copy the result into Src1/Src2 fields.

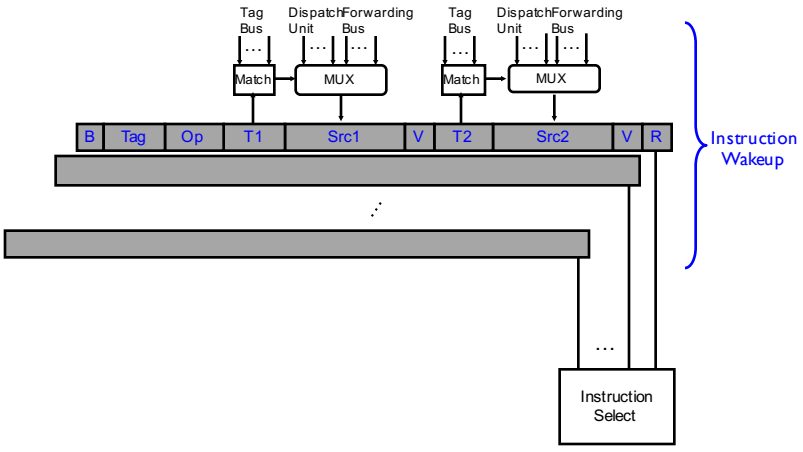


Commit Stage

- If Dst of the entry at the front of the ROB is valid (i.e., instruction has completed), write it to the destination register D. Remove this ROB entry by clearing B.
- Up to *n* instructions can be committed per cycle.



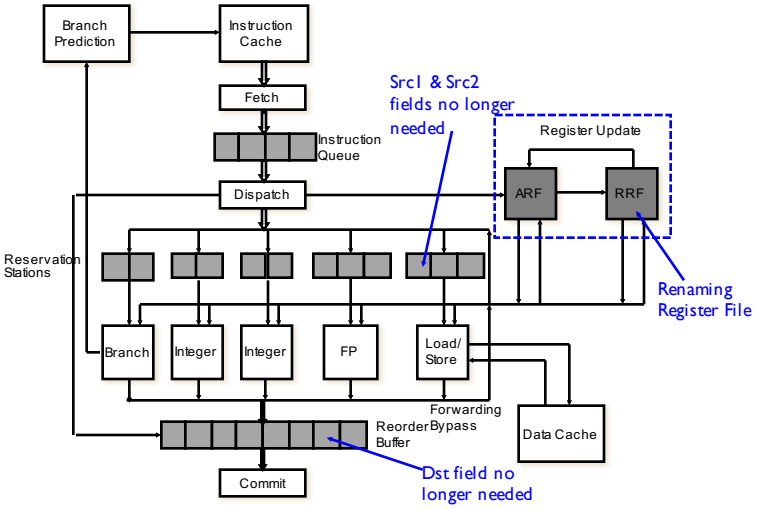
Reservation Stations



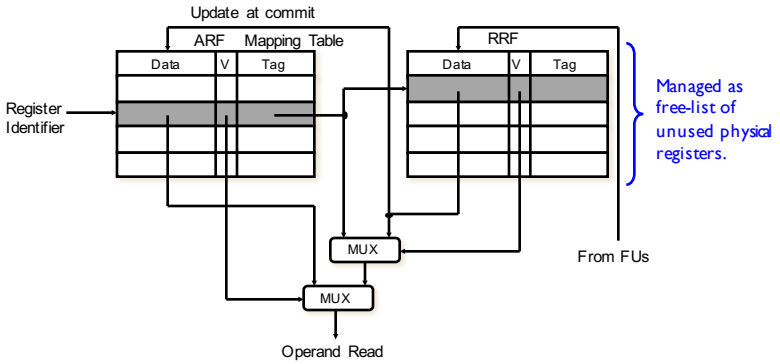
Register Renaming

- Register renaming in our example thus far has been based on ROB:
  - Dst field in ROB acts as a renaming register.
  - Tag field in RF points to the renaming register in ROB.
  - A renaming register is allocated for every instruction in flight.
- However, not all instructions require a renaming register:
  - e.g., Branch instructions
  - May be wasteful.
- Some processor use a separate register file for renaming:
  - Architected Register File (ARF) - logical registers
  - Renaming Register File (RRF) - physical registers
  - ROB no longer has Dst field for data
  - Data will be held in one of the renaming register in RRF
  - Data from FUs will be forwarded and written to renaming registers in RRF.

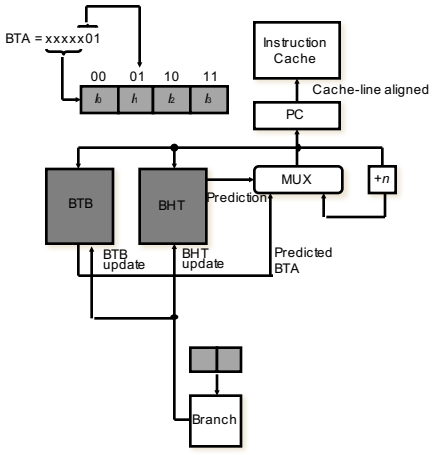
Renaming Register File (1)



Renaming Register File (2)



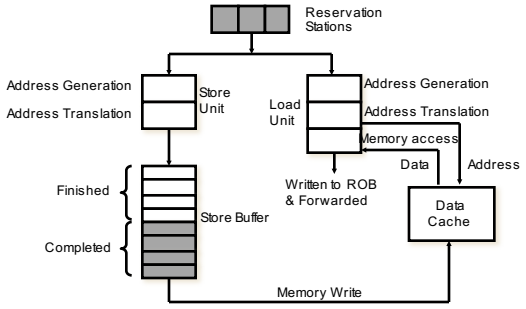
Branch Prediction



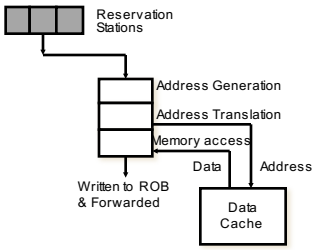
- BTB and BHT accessed during Fetch stage.
- Instruction fetched is cache aligned.
- Predicted BTA will be the first branch in fetched group of instructions.
- When branch is resolved
  - BHT is updated (if needed).
  - BTB entry is either left alone (taken) or deleted (not taken).
- Can speculate up to the number of RSs available.
  - After branch is resolved, speculative branch instructions are either made non-speculative or invalidated.
  - If branch is invalidated, ROB entries occupied by the branch and its succeeding insts. are all deallocated.

Memory Dataflow

- Assumes loads/stores are issued in-order from RSs.

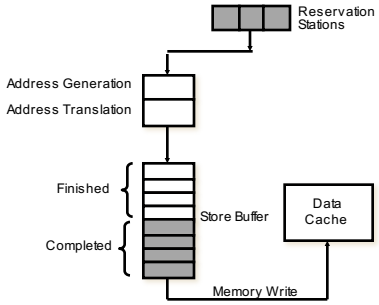


Loads



- When rs of lw rt, disp(rs) is available, inst. is issued.
- Address = disp + [rs]
- Virtual to Physical address translation (TLB).
- Data Cache accessed and data written to ROB and forwarded.
- When load instruction is at the head of ROB, RF is updated (i.e., committed).

Stores

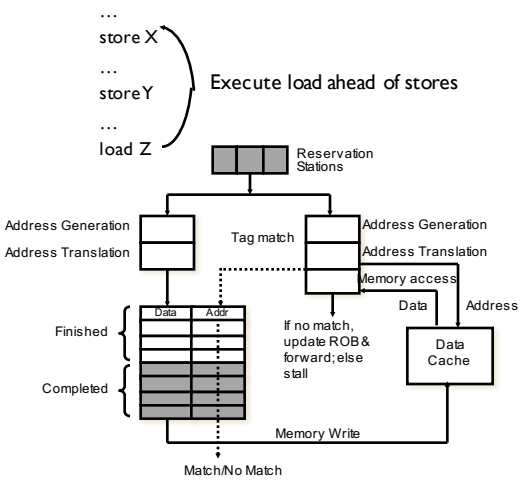


Stores are processed differently:

- When both rs and rt of sw rt,disp(rs) is available, inst. is issued.
- Address = disp + [rs]
- Virtual to Physical address translation (TLB).
- Address and data is put into Store Buffer and is considered *finished*.
- When store instruction is at the head of ROB, it is considered *completed*.
- When *memory bus is available*, completed store exists the store buffer and memory is updated (i.e., committed).

Load Bypassing Stores

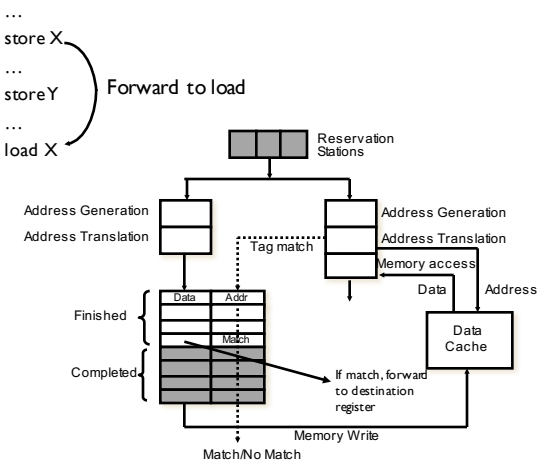
- Why have store buffers but no load buffers? Consider the following:



- If no match, update ROB and forward.
- If match
  - stall
  - or discard loaded data & load is held back in RS

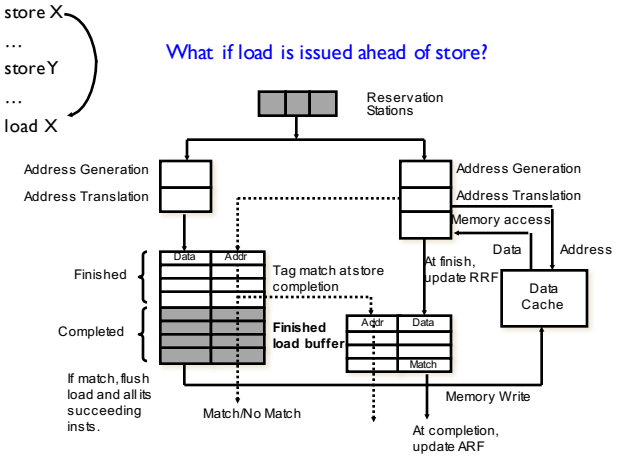
Load Forwarding

- If load depends on a previous store…



OoO Loads and Stores

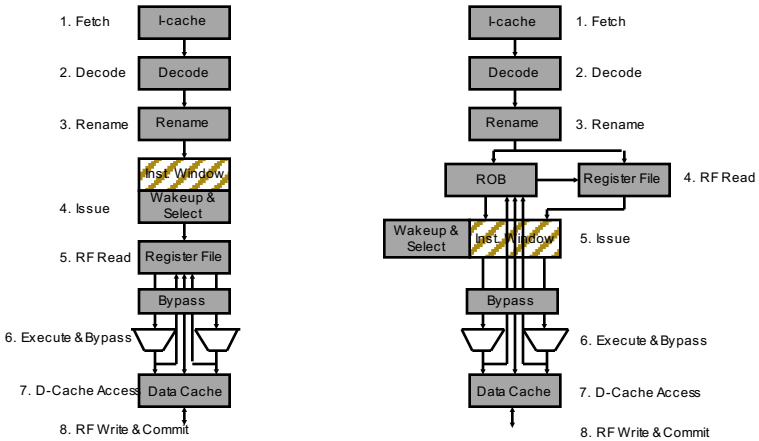
- We need to be careful since loads and stores can be issued OoO from the RSs.



Complexity of SuperScalar Processors

Complexity of SS Processors

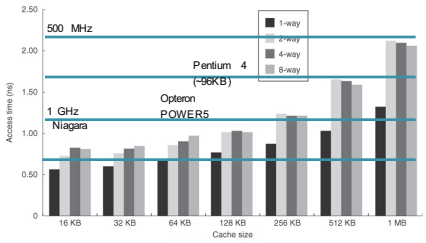
Primary Delay Structures



MIPS, Pentium 4, Alpha, UltraSparc

PowerPC, Pentium II & III, Core

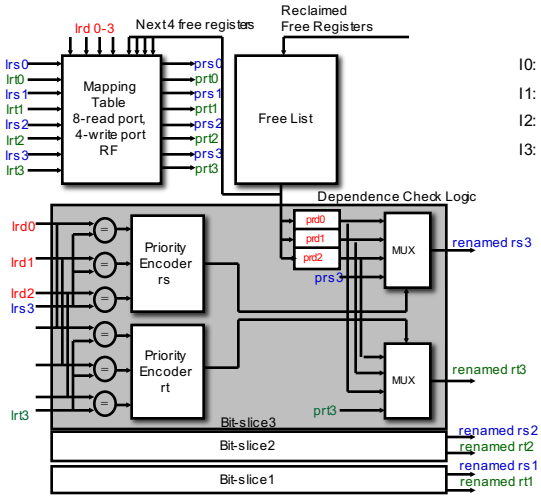
Fetch Stage

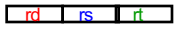


Based on CACTI, 90 nm, 64-byte block. 1.32, 1.39, & 1.43 for 2-way, 4-way, and 8-way relative to direct-mapped.

- Current generation of processors provide 2+ GHz clock rate.
- Most cache structures cannot provide instruction (or data) in one clock cycle!
  - AMD Opteron: 2 cycles @ 2.8 GHz
  - Intel Pentium 4: 4 cycles @ 3.6 GHz (Trace cache).
  - IBM Power 5: 1 cycle @ 2 GHz (0.13 μm)
  - Sun Niagara: 1 cycle @ 1.2 GHz

Rename Stage





I0: op R1,R2,R3
I1: op R4,R1,R5
I2: op R1,R6,R7
I3: op R6,R3,R1

Log.	Phy.
0	12
1	13=>0=>2
2	15
3	14
4	9=>1
5	7
6	6=>3
7	8

Free pool: 0, 1, 2, 3, 4, 5, 10, 11

Step 1	Step 2
op P0,P15,P14	op P0,P15,P14
op P1,P13,P7	op P1,P0,P7
op P2,P6,P8	op P2,P6,P8
op P3,P14,P13	op P3,P14,P2

Delay Analysis of Rename Stage

The critical path is the time it takes to read the Mapping Table (register file). Delay through the output MUX is minimal.

$$T_{rename} \propto pR^{1/2}$$

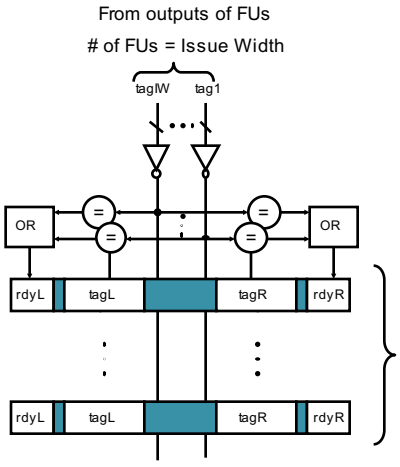
where

p - number of ports

R - number of entries

Usually, R is small (≤ 32), thus rename delay is not critical.

Wakeup Logic



- Responsible for determining when instructions are ready to be issued.

Tag1. Add R1,R2,R3

Tag2. Add R4,R1$_{Tag1}$,R2

Tag3. Add R5,R2,R1$_{Tag1}$

Delay Analysis of Wakeup Logic

Delay consists of buffers to drive the tag bits, the comparators to match tags, and OR the individual match signals:

$$T_{wakeup} = T_{tagdrive} + T_{tagmatch} + T_{matchOR}$$

$T_{tagdrive}$ depends on the length of tag lines and the number of comparators on the tag lines (both depend on WINSIZE).

$$T_{wakeup} = c_0 + (c_1 + c_2 \times IW) \times WINSIZE + (c_3 + c_4 \times IW) + c_5 \times IW^2) \times WINSIZE^2$$

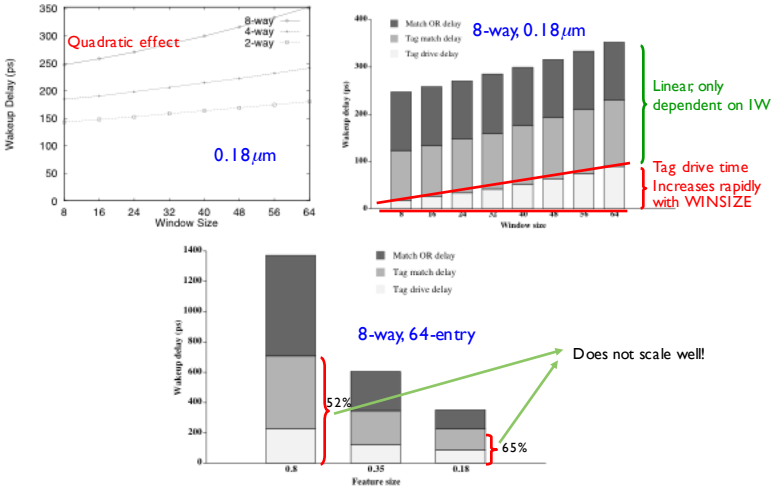
where

IW - issue width

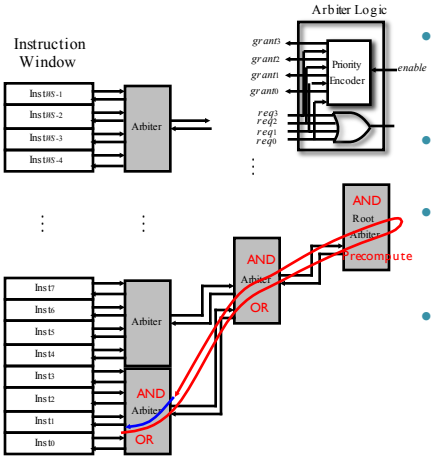
WINSIZE - window size

$c_{0\text{-}5}$ - fixed constants for given technology & ISA

Delay Analysis of Wakeup Logic



Select Logic

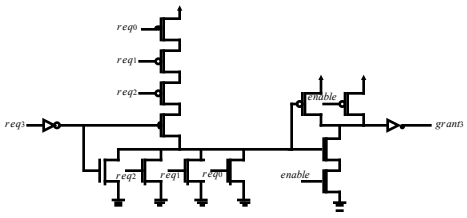


- Select logic chooses instructions from the pool of ready instructions in the IW.
- # of ready instructions > # and types of FUs.
- For different types of FUs, a FU tag accompanies request and grant signals.
- The priority is given to the head entries. Assumes no compaction of the issue window.

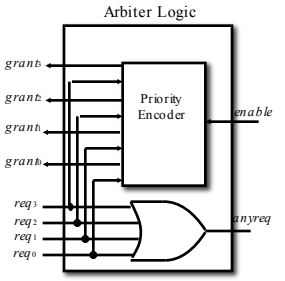
$grant = 2 \times \lceil \log_4(WS) - 1 \rceil$ gds

For WS=Window Size =32, $grant$ = 6 gds

Priority Encoder



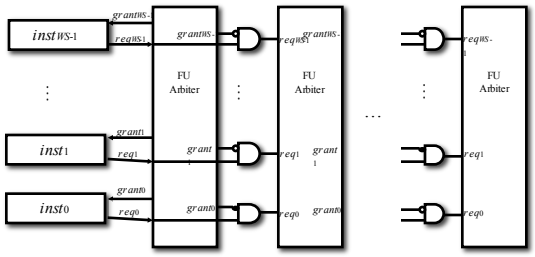
$grant_3 = enable \cdot (req_0' \cdot req_1' \cdot req_2' \cdot req_3)$



$grant_0 = enable \cdot (req_0)$
$grant_1 = enable \cdot (req_0' \cdot req_1)$
$grant_2 = enable \cdot (req_0' \cdot req_1' \cdot req_2)$
$grant_3 = enable \cdot (req_0' \cdot req_1' \cdot req_2' \cdot req_3)$

ANDed with enable | Precompute

Delay Analysis of Selection Logic



- Can't have too many FUs.
- PowerPC has separate RS for each FU.
- MIPS and Alpha have separate integer and FP IQs.
  - MIPS: 3 IQs for INT, FP, and Load/Store
  - Alpha; 2 IQs for INT and FP
  - Pentium 4: 2 IQs for INT/FP and Load/Store

Register File Access

- Register file access time depends on number of *ports* and *entries*.
  - *n*-way issue requires at most 2*n* read ports and *n* write ports
- RF design
  - Separate physical and logical RFs
  - Combined physical and logical RF
- To support large instruction window and number of in-flight instructions, a physical RF with many entries and ports is required.
  - Pentium 4 has 128 reaming registers.
  - POWER 5 is a 8-way issue.

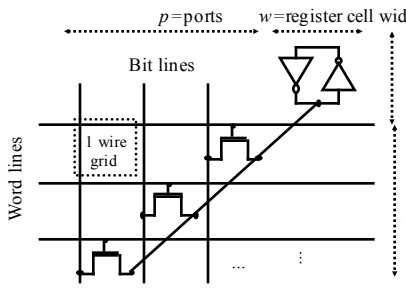
Register File Delay Model

transistor gate capacitance, wire capacitance, wire delay

$$T_{access}(R, p) = \log_4((C_{word}+(w+p)C_w)bR)+t_w(w+p)(bR)^{1/2}+$$

$$\log_4((C_{bit}+(h+p)C_w)R)+t_w(h+p)(bR)^{1/2},$$

transistor gate capacitance; wire capacitance; wire delay



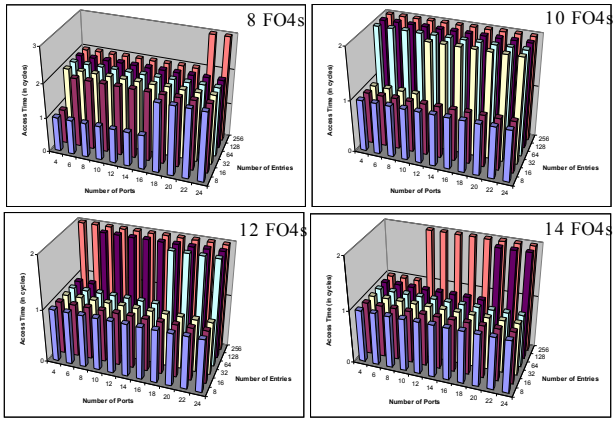
C_{bit} = bit line transistor capacitance
C_{word} = word line transistor capacitance
C_w = one track of wire capacitance
t_w = wire delay
R = # of registers
b = data width

In terms of fanout-of-4 (FO4) delays:
FO4 = 360 L_{drawn} (picoseconds),
L_{drawn} = minimum gate length

Register File Delay

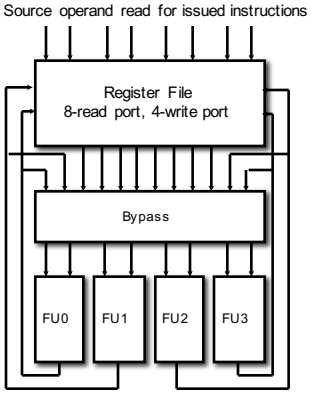
90nm process, 8-14 FO4s => 3.8~2.2 GHz clock

1.8 FO4 overhead for skew, jitter, and latch times



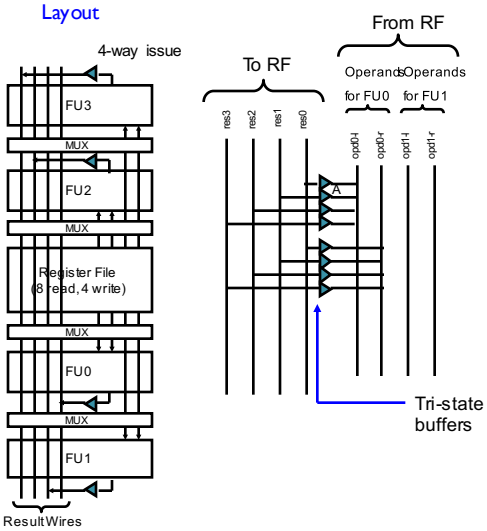
Complexity of Bypass Logic

Forwards result values from completed instruction to dependent instruction, bypassing the RF.



Complexity of Bypass Logic

Layout



To bypass result from FU1 (res1) to the left input of FU0 (opd0-l), tri-state buffer A is turned on. Otherwise, operand come from RF read ports.

Delay Analysis of Bypass Logic

Delay is largely determined by the time it takes for the driver at the output of each FU to drive the result value on the result wire, which in turn depends on the length of the result wires.

$$T_{bypass} = 0.5 \times R_{metal} \times C_{metal} \times L^2$$

where

L is the length of result wires

R_{metal} is the resistance per unit length

C_{metal} is the capacitance per unit length

Effect:

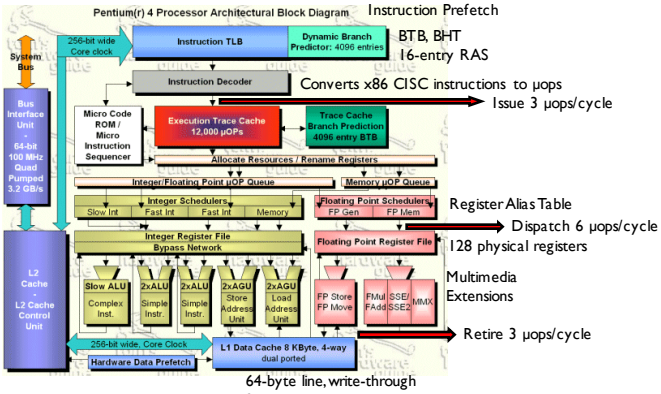
- Increasing IW increases # of FUs, which increases L (distance from RF to FUs).
- Increasing IW also increase the fan-in of the operand MUXes connected to a given result wire, which increases the capacitance of the result wires.

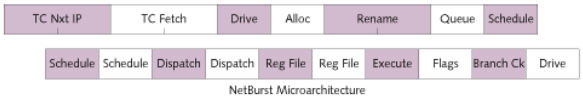
Case Study: Pentium 4

Case Study: Pentium 4

Pentium 4 Willamette
Introduced November, 2000

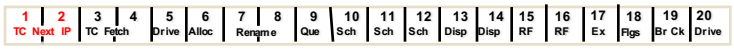
NetBurst Architecture



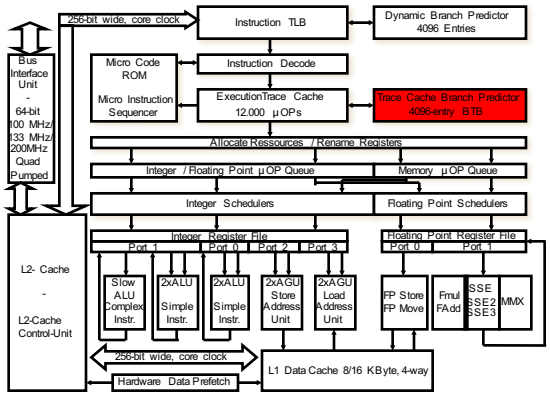


NetBurst Microarchitecture

Pentium 4: TC Next IP



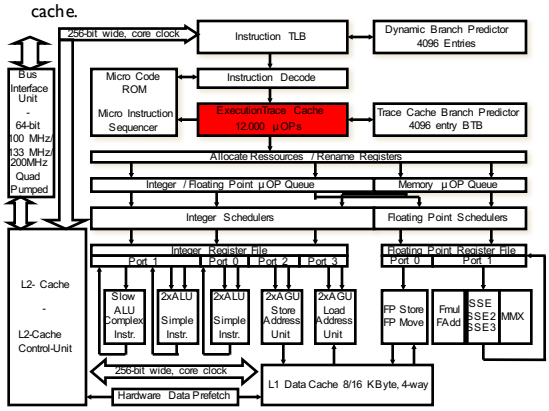
Trace Cache Next Instruction Pointer - Obtain pointer from BTB, indicating the location of next instruction.



Pentium 4: TC Fetch

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Next IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

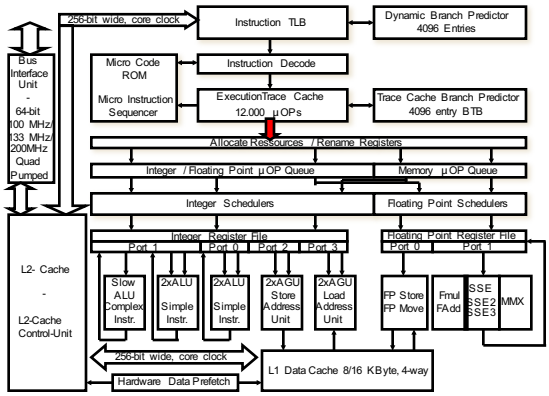
Trace Cache Fetch - Read the decoded instructions (µOPs) out of the execution trace cache.



Pentium 4: Drive

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Next IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

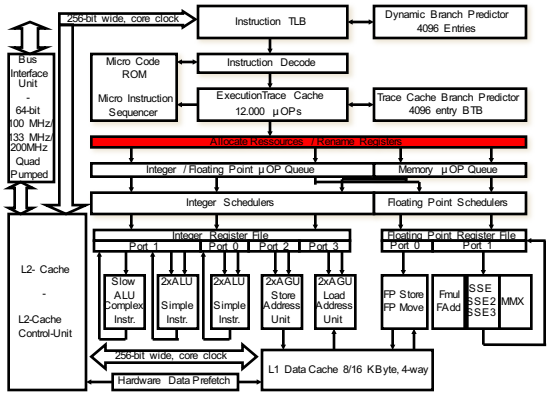
Wire Delay - Drive the µOPs to the allocator.



Pentium 4: Allocate

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Next IP		TC Fetch		Drive	Alloc		Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

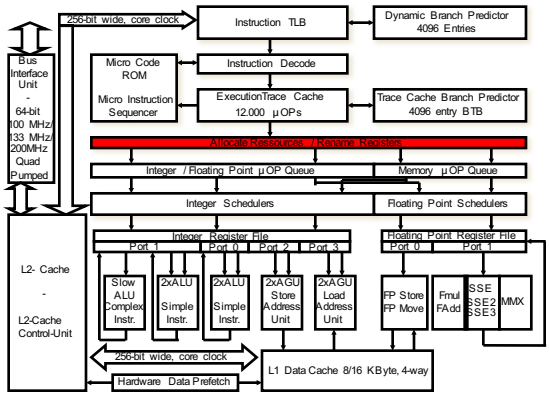
Allocate Resources - The resources include load buffers, store buffers, registers, reorder buffers (to track the status of 126 µOPs), etc.



Pentium 4: Rename

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Next IP		TC Fetch		Drive	Alloc		Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

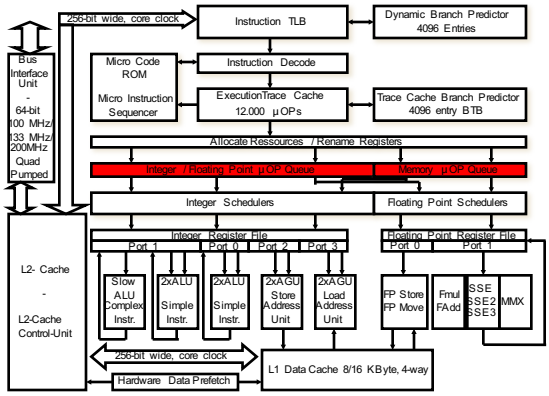
Register Renaming - Rename the logical registers to the physical register space (128 physical registers are implemented).



Pentium 4: Queue

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Next IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

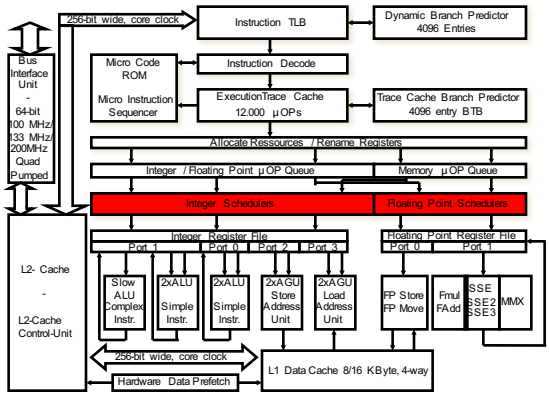
Write into the µOP Queue - µOPs are placed into the queues, where they are held until there is room in the scheduler.



Pentium 4: Schedule

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Next IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

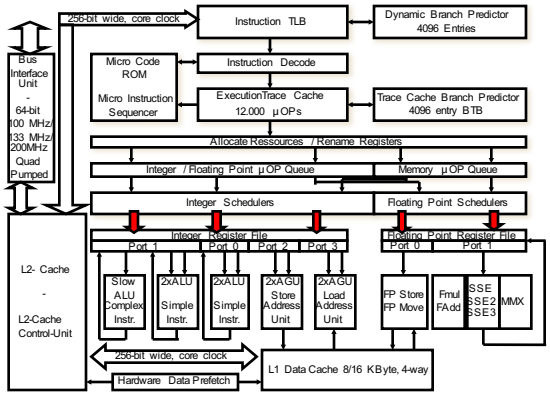
Schedule - Write into the schedulers and compute dependencies. Watch for dependency to resolve.



Pentium 4: Dispatch

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Next IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

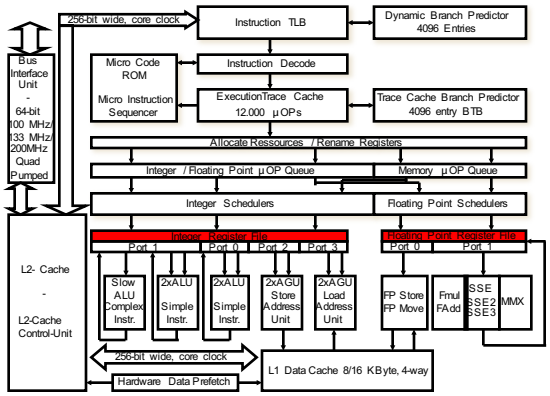
Dispatch - Send the µOPs to the appropriate FUs.



Pentium 4: Register File

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Next IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

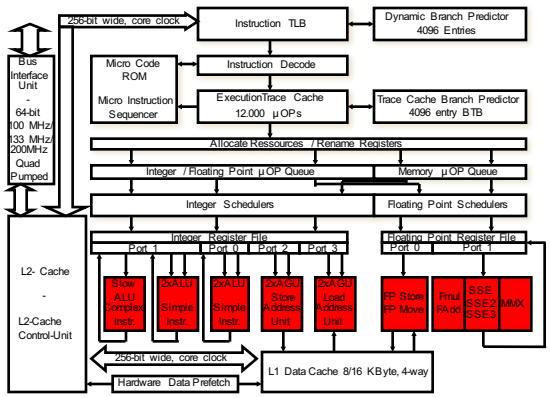
Register File - Read the register file. These are the sources for the pending operation (ALU or other).



Pentium 4: Execute

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Next IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Figs	Br Ck	Drive

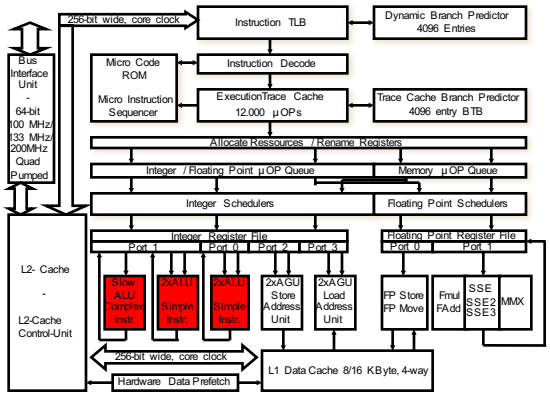
Execute - Execute the µOPs on the appropriate FUs.



Pentium 4: Flags

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Next IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Figs	Br Ck	Drive

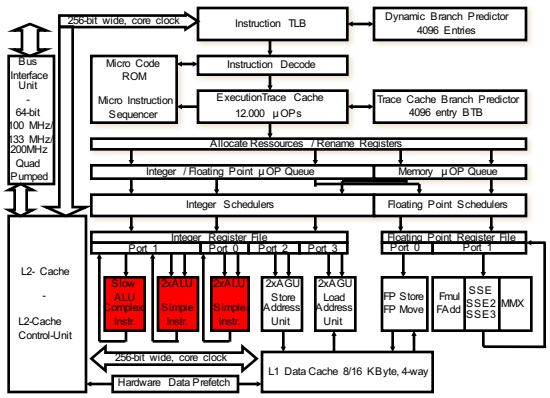
Compute Flags - These are typically input to a branch instruction.



Pentium 4: Branch Check

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Next IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

Branch Check - The branch operation compares the result of the actual branch direction with the prediction.



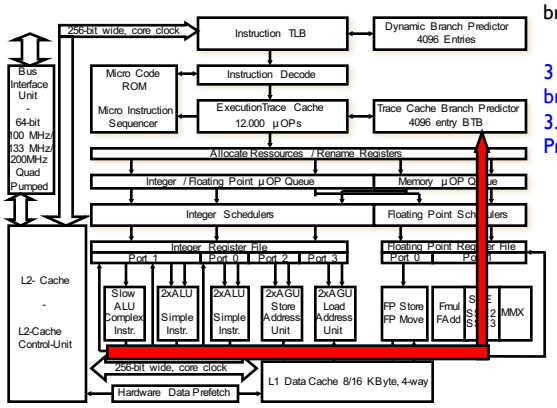
Pentium 4: Drive

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Next IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

Wire Delay - Drive the result of the branch check to the front end of the machine.

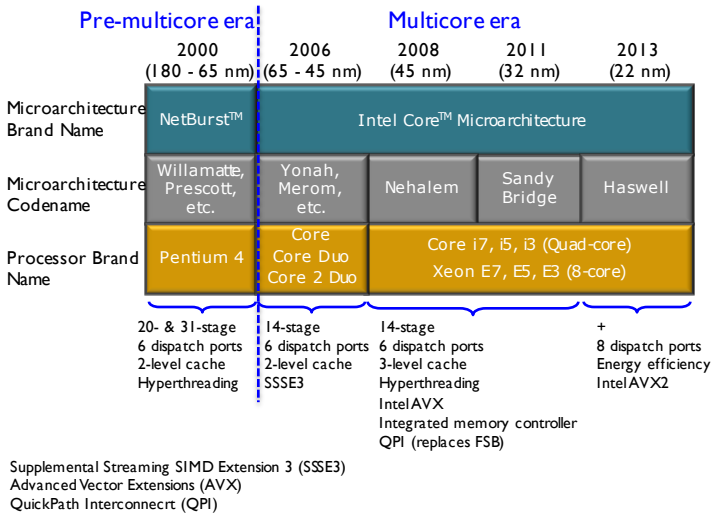
20-cycle branch penalty!

31-cycle branch penalty for 3.8 GHz Pentium Prescot!

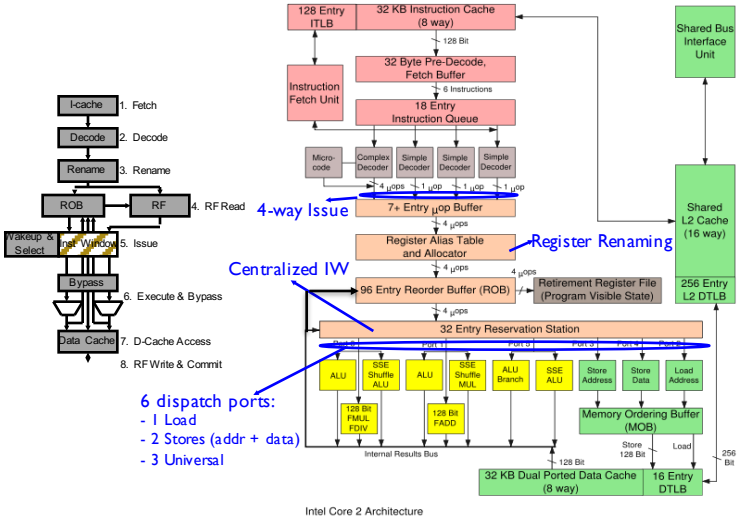


The Core™ Microarchitecture

Evolution of Intel Architectures

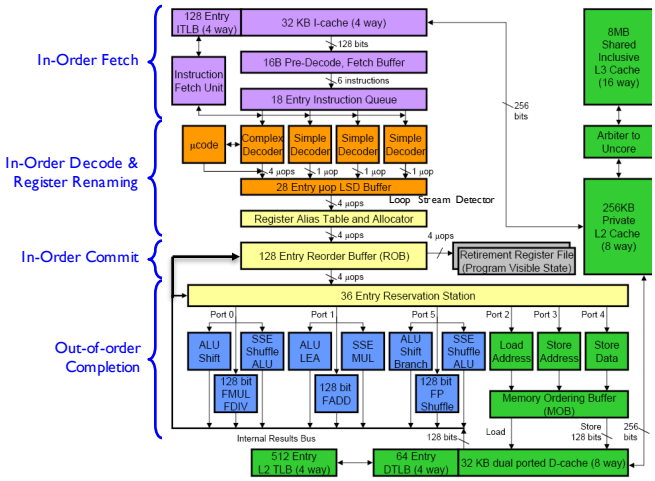


The Core 2 Microarchitecture



Intel Core 2 Architecture

Nehalem



Haswell

- Evolutionary improvements to Nehalem:
  - Power efficiency
  - AVX2
  - Integrated graphics unit
  - Larger L2 cache
  - Level 3 cache
  - 8 Dispatch ports (2 more than previous models)
  - More buffers

Buffer Size Comparison

	Nehalem	Sandy Bridge	Haswell
ROB Entries	128	168	192
Load Buffer Entries	48	64	72
Store Buffer Entries	32	36	42
RS Entries	36	54	60
Integer RRF	N/A	160	168
FP RRF	N/A	144	168
µop Decode Queue	28/thread	28/thread	56

Questions?