Computer Structure

Intel® Core™ \(\mu\)Arch

Lihu Rappoport and Adi Yoaz
## Tick/Tock Development Model

<table>
<thead>
<tr>
<th>Year</th>
<th>Generation</th>
<th>Intel® μArch</th>
</tr>
</thead>
<tbody>
<tr>
<td>2006</td>
<td>1st Generation</td>
<td>Core2™ Duo (Merom)</td>
</tr>
<tr>
<td>2007</td>
<td>1st Generation</td>
<td>Core2™ Duo (Merom)</td>
</tr>
<tr>
<td>2008</td>
<td>1st Generation</td>
<td>Core™ (Nehalem)</td>
</tr>
<tr>
<td>2009</td>
<td>2nd Generation</td>
<td>Core™ (Nehalem)</td>
</tr>
<tr>
<td>2010</td>
<td>3rd Generation</td>
<td>Core™ (Sandy Bridge)</td>
</tr>
<tr>
<td>2011</td>
<td>3rd Generation</td>
<td>Core™ (Sandy Bridge)</td>
</tr>
<tr>
<td>2012</td>
<td>4th Generation</td>
<td>Core™ (Haswell)</td>
</tr>
<tr>
<td>2013</td>
<td>4th Generation</td>
<td>Core™ (Haswell)</td>
</tr>
<tr>
<td>2014</td>
<td>5th Generation</td>
<td>Core™ (Skylake)</td>
</tr>
<tr>
<td>2015</td>
<td>5th Generation</td>
<td>Core™ (Skylake)</td>
</tr>
<tr>
<td>2016</td>
<td>6th Generation</td>
<td>Core™ (Skylake)</td>
</tr>
</tbody>
</table>

**Architectures:**
- **Merom**
- **Penryn**
- **Nehalem**
- **Westmere**
- **Sandy Bridge**
- **Ivy Bridge**
- **Haswell**
- **Brodwell**
- **Skylake**

**Technology Nodes:**
- **65nm**
- **45nm**
- **32nm**
- **22nm**
- **14nm**
6th Generation Intel Core™ – Skylake

- 14nm Process Technology
- Quad core die, with Intel HD Graphics 530
Core at a Glance

Next generation branch prediction
- Improves performance and saves wasted work

Improved front-end
- Initiate TLB and cache misses speculatively
- Handle cache misses in parallel to hide latency
- Leverages improved branch prediction

Deeper buffers
- Extract more instruction parallelism
- More resources when running a single thread

More execution units, shorter latencies
- Power down when not in use

More load/store bandwidth
- Better prefetching, better cache line split latency & throughput, double L2 bandwidth
- New modes save power without losing performance

No pipeline growth
- Same branch misprediction latency
- Same L1/L2 cache latency
• Predict branch targets
  – Direct Calls and Jumps – target provided by a Target Array
  – Indirect Calls and Jumps – predicted either as having a fixed target or as having targets that vary based on execution path
  – Returns – predicted by a 16 entry Return Stack Buffer (RSB)

• For conditional branches
  – predict if taken or not

• The BPU makes predictions for 32 bytes at a time
  – Twice the width of the fetch engine
  – Enables taken branches to be predicted with no penalty
Instruction TLB

- **Initiate TLB and cache misses speculatively**
  - Handle cache misses in parallel to hide latency
  - Leverages improved branch prediction
Instruction Fetch and Pre-decode

- **32KB 8-way I-Cache**
  - Fetches aligned 16 bytes per cycle
    - Typical programs average ~4 bytes per instruction
  - A misaligned target into the line or a taken branch out of the line reduces the effective number of instruction bytes fetched
    - In typical integer code: a taken branches every ~10 instructions, which translates into a partial fetch every 3~4 cycles

- **The Pre-Decode Unit**
  - Determines the length of each instruction
  - Decodes all prefixes associated with instructions
  - Mark various properties of instructions for the decoders
    - for example, “is branch”
Instruction Pre-decode and IQ

- **The pre-decode unit writes ≤6 instructions/cycle into the IQ**
  - If a fetch line contains >6 instructions
    - Continues to pre-decode ≤6 instructions/cycle, until all instructions in the fetch line are written into IQ
    - The subsequent fetch line starts pre-decode only after the current fetch completes
  - A fetch of line of 7 instructions takes 2 cycles to pre-decode
    - Average of 3.5 inst/cycle (still higher than IPC of most apps)

- **Length changing prefixes (LCP) change the inst. length**
  - Each LCP within the fetch line takes 3 cycle to pre-decode

- **The Instruction Queue is 18 instructions deep**
Micro Operations (Uops)

- Each X86 inst. is broken into one or more RISC μ-ops
  - Each μop is (relatively) simple

- Simple instructions translate to a few μops
  - Typical μop count (it is not necessarily cycle count!)
    - Reg-Reg ALU/Mov inst: 1 μop
    - Mem-Reg Mov (load) 1 μop
    - Mem-Reg ALU (load + op) 2 μops
    - Reg-Mem Mov (store) 2 μops (st addr, st data)
    - Reg-Mem ALU (ld + op + st) 4 μops
**Instruction Decode and μop Queue**

- **4 Decoders**, which decode instructions into μops
  - 1\(^{st}\) decoder can decode all instructions of up to 4 μops
  - The other 3 decoders handle common single μop instructions

- **Instructions with >4 μops generate μops from the MSROM**

- **Up to 4 μops/cycle are delivered into the μop Queue**
  - Buffers 56 μops (28 μops / thread when running 2 threads)
  - Decouples the front end and the out-of-order engine
  - Helps hiding bubbles introduced between the various sources of μops in the front end
• **The IQ sends up to 5 inst. / cycle to the decoders**

• **Merge two adjacent instructions into a single μop**
  - A macro-fused instruction executes with a single dispatch
    - Reduces latency and frees execution resources
    - Increased decode, rename and retire bandwidth
    - Power savings from representing more work in fewer bits

• **The 1\textsuperscript{st} instruction modifies flags**
  - CMP, TEST, ADD, SUB, AND, INC, DEC

• **The 2\textsuperscript{nd} instruction pair is a conditional branch**

• **These pairs are common in many types of applications**
Stack Pointer Tracker

- **PUSH, POP, CALL, RET implicitly update ESP**
  - Add or sub an offset, which would require a dedicated \(\mu\)op
  - The Stack Pointer Tracker performs implicit ESP updates

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ESP Offset</th>
<th>(\Delta)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH EAX</td>
<td>ESP - 4</td>
<td>(\Delta = \Delta - 4)</td>
</tr>
<tr>
<td></td>
<td>STORE [ESP], EAX</td>
<td>STORE [ESP-4], EAX</td>
</tr>
<tr>
<td>PUSH EBX</td>
<td>ESP = ESP - 4</td>
<td>(\Delta = \Delta - 4)</td>
</tr>
<tr>
<td></td>
<td>STORE [ESP], EBX</td>
<td>STORE [ESP-8], EBX</td>
</tr>
<tr>
<td>INC ESP</td>
<td>ESP = ADD ESP, 1</td>
<td>(\Delta = 0)</td>
</tr>
</tbody>
</table>

- **Provides the following benefits**
  - Improves decode BW: PUSH, POP and RET become single \(\mu\)op instructions
  - Conserves rename, execution and retire resources
  - Remove dependencies on ESP – can execute stack operations in parallel
  - Saves power
• **Micro-Fusion**

  - **Fuse multiple μops from same instruction into a single μop**
  - Instruction which decode into a single micro-fused μop can be handled by all decoders
  - Improves instruction bandwidth delivered from decode to retirement and saves power
  - A micro-fused μop is dispatched multiple times in the OOO
    - As it would if it were not micro-fused

• **Micro-fused instructions**

  - Stores are comprised of 2 μop: store-address and store-data
    - Fused into a single μop
  - Load + op instruction
    - e.g., FADD DOUBLE PTR [RDI+RSI*8]
  - Load + jump
    - e.g., JMP [RDI+200]

From the Optimization Manual
Decoded μop-Cache

- **Caches the μops coming out of the decoders**
  - Up to 1.5K μops (32 sets × 8 ways × 6 μops/way)
  - Next time μops are taken from the μop Cache
  - ~80% hit rate for most applications
  - Included in the IC and iTLB, flushed on a context switch

- **Higher Bandwidth and Lower Latency**
  - More cycles sustaining 4 instruction/cycle
    - In each cycle provide μops for instructions mapped to 32 bytes
    - Able to ‘stitch’ across taken branches in the control flow
Decoded μop-Cache

- Decoded μop Cache lets the normal front end sleep
  - Decode one time instead of many times

- Branch misprediction penalty reduced
  - The correct path is also the most efficient path

Save Power while Increasing Performance
Loop Stream Detector (LSD)

- **LSD detects small loops that fit in the μop queue**
  - The μop queue streams the loop, allowing the front-end to sleep
  - Until a branch miss-prediction inevitably ends it

- **Loops qualify for LSD replay if all following conditions are met**
  - Up to 28 μops, with \( \leq 8 \) taken branches, \( \leq 8 \) 32-byte chunks
  - All μops are also resident in the μop-Cache
  - No CALL or RET
  - No mismatched stack operations (e.g., more PUSH than POP)
The Renamer

- **Moves ≤4µops/cycle from the µop-queue to the OOO**
  - Renames architectural sources and destinations of the µops to micro-architectural sources and destinations
  - Allocates resources to the µops, e.g., load or store buffers
  - Binds the µop to an appropriate dispatch port
  - Up to 2 branches each cycle

- **Some µops are executed to completion during rename, effectively costing no execution bandwidth**
  - A subset of register-to-register MOV and FXCHG
  - Zero-Idioms
  - NOP
Dependency Breaking Idioms

- **Zero-Idiom** – an instruction that zeroes a register
  - Regardless of the input data, the output data is always 0
  - E.g.: XOR REG, REG and SUB REG, REG
  - No μop dependency on its sources

- **Zero-Idioms are detected and removed by the Renamer**
  - Do not consume execution resource, have zero exe latency

- **Zero-Idioms remove partial register dependencies**
  - Improve instruction parallelism

- **Ones-Idiom** – an instruction that sets a register to “all 1s”
  - Regardless of the input data the output is always "all 1s"
  - E.g., CMPEQ XMM1, XMM1;
    - No μop dependency on its sources, as with the zero idiom
    - Can execute as soon as it finds a free execution port
  - As opposed to Zero-Idiom, the Ones-Idiom μop must execute
Out-of-Order Cluster

- **The Scheduler queues μops until all source operands are ready**
  - Schedules and dispatches ready μops to the available execution units in as close to a first in first out (FIFO) order as possible

- **Physical Register File (PRF)**
  - Instead of centralized Retirement Register File
    - Single copy of every data with no movement after calculation
  - Allows significant increase in buffer sizes
    - Dataflow window ~33% larger

- **Retirement**
  - Retires μops in order and handles faults and exceptions
Intel® Advanced Vector Extensions

- **Vectors are a natural data-type for many apps**
  - Extend SSE FP instruction set to 256 bits operand size
  - Extend all 16 XMM registers to 256bits

- **New, non-destructive source syntax**
  - VADDPS ymm1, ymm2, ymm3

- **New Operations to enhance vectorization**
  - Broadcasts, masked load & store

Wide vectors+ non-destructive source: more work with fewer instructions
Extending the existing state is area and power efficient
Haswell AVX2: FMA & Peak FLOPS

- 2 new FMA units provide 2× peak FLOPs/cycle
  - Fused Multiply-Add

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>SP/cyc</th>
<th>DP/cyc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>SandyBridge</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Haswell</td>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>

- 2× cache bandwidth to feed wide vector units
  - 32-byte load/store for L1
  - 2× L2 bandwidth

- 5-cycle FMA latency same as an FP multiply

<table>
<thead>
<tr>
<th>Latency (clks)</th>
<th>Prior Gen</th>
<th>Haswell</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>MulPS, PD</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>AddPS, PD</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Mul+Add /FMA</td>
<td>8</td>
<td>5</td>
<td>1.6</td>
</tr>
</tbody>
</table>
Haswell Execution Units

Unified Reservation Station

Port 0
- Integer ALU & Shift
- FMA
- FP Multiply
- Vector Int Multiply
- Vector Logicals
- Branch
- Divide
- Vector Shifts

Port 1
- Integer ALU & LEA
- FMA FP Mult
- FP Add
- Vector Int ALU
- Vector Logicals

Port 2
- Load & Store Address
- Port 0

Port 3
- Store Data
- Port 1

Port 4
- Integer ALU & LEA
- Port 2

Port 5
- Integer ALU & Shift
- Port 3

Port 6
- Integer ALU & LEA
- Port 4

Port 7
- Store Address
- Port 5

2×FMA
- Doubles peak FLOPs
- Two FP multiplies benefits legacy

4th ALU
- Great for integer workloads
- Frees Port0 & 1 for vector

New Branch Unit
- Reduces Port0 Conflicts
- 2nd EU for high branch code

New AGU for Stores
- Leaves Port 2 & 3 open for Loads

Intel® Microarchitecture (Haswell)
The L1 D$ handles 2× 256-bit loads + 1× 256-bit store per cycle

The ML$ can service one cache line (64 bytes) each cycle

72 load buffers keep load μops from allocation till retirement
  - Re-dispatch blocked loads

42 store buffers keep store μops from allocation till the store value is written to L1-D$
  - or written to the line fill buffers – for non-temporal stores
L1 Data Cache

- Handles 2× 256-bit loads + 1× 256-bit store per cycle

- Maintains requests which cannot be completed
  - Cache misses
  - Unaligned access that splits across cache lines
  - Data not ready to be forwarded from a preceding store
  - Load block due to cache line replacement

- Handles up to 10 outstanding cache misses in the LFBs
  - Continues to service incoming stores and loads

- The L1 D$ is a write-back write-allocate cache
  - Stores that hit in the L1-D$ do not update the L2/LLC/Mem
  - Stores that miss the L1-D$ allocate a cache line
Stores to memory are executed in two phases

**At EXE**
- Fill store buffers with linear + physical address and with data
- Once store address and data are known
  - Forward Store data to load operations that need it

**After the store retires – completion phase**
- First, the line must be in L1 D$, in E or M MESI state
  - Otherwise, fetch it using a *Read for Ownership* request from
    - L1 D$ → L2$ → LLC → L2$ and L1 D$ in other cores → Memory
- Read the data from the store buffers and write it to L1 D$ in M state
  - Done at retirement to preserve the order of memory writes
- Release the store buffer entry taken by the store
  - Affects performance only if store buffer becomes full (allocation stalls)
  - Loads needing the store data get when store is executed
Store Forwarding

• **Forward data directly from a store to a load that needs it**
  - Instead of having the store write the data to the DCU and then the load read the data from the DCU

• **Store to load forwarding conditions**
  - The store is the last store to that address, prior to the load
    - Requires addresses of all stores older than the load to be known
  - The store contains all data being loaded
  - The load is from a write-back memory type
  - Neither the load nor the store are non-temporal accesses
  - The load is not a 4 an 8 byte load that crosses 8 byte boundary, relative to the preceding 16- or 32-byte store
  - The load does not cross a 16-byte boundary of a 32-byte store

From the Optimization Manual
Memory Disambiguation

- **A load may depend on a preceding store**
  - A load is blocked until all preceding store addresses are known

- **Predict which loads do not depend on any previous stores**
  - Forward data from a store or L1 D$ even when not all previous store addresses are known
  - The prediction is verified, and if a conflict is detected
    - The load and all succeeding instructions are re-fetched
  - Always assumes dependency between loads and earlier stores that have the same address bits 0:11

- **The following loads are not disambiguated**
  - Loads that cross the 16-byte boundary
  - 32-byte Intel AVX loads that are not 32-byte aligned
Data Prefetching

- Two hardware prefetchers load data to the L1 D$:
  - DCU prefetcher (the streaming prefetcher)
    - Triggered by an ascending access to very recently loaded data
    - Fetches the next line, assuming a streaming load
  - Instruction pointer (IP)-based stride prefetcher
    - Tracks individual load instructions, detecting a regular stride
      - Prefetch address = current address + stride
      - Detects strides of up to 2K bytes, both forward and backward
### Core Cache Size/Latency/Bandwidth

<table>
<thead>
<tr>
<th>Metric</th>
<th>Nehalem</th>
<th>Sandy Bridge</th>
<th>Haswell</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Instruction Cache</td>
<td>32K, 4-way</td>
<td>32K, 8-way</td>
<td>32K, 8-way</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>32K, 8-way</td>
<td>32K, 8-way</td>
<td>32K, 8-way</td>
</tr>
<tr>
<td>Fastest Load-to-use</td>
<td>4 cycles</td>
<td>4 cycles</td>
<td>4 cycles</td>
</tr>
<tr>
<td>Load bandwidth</td>
<td>16 Bytes/cycle</td>
<td>32 Bytes/cycle (banked)</td>
<td>64 Bytes/cycle</td>
</tr>
<tr>
<td>Store bandwidth</td>
<td>16 Bytes/cycle</td>
<td>16 Bytes/cycle</td>
<td>32 Bytes/cycle</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>256K, 8-way</td>
<td>256K, 8-way</td>
<td>256K, 8-way</td>
</tr>
<tr>
<td>Fastest load-to-use</td>
<td>10 cycles</td>
<td>11 cycles</td>
<td>11 cycles</td>
</tr>
<tr>
<td>Bandwidth to L1</td>
<td>32 Bytes/cycle</td>
<td>32 Bytes/cycle</td>
<td>64 Bytes/cycle</td>
</tr>
<tr>
<td>L1 Data TLB</td>
<td>4K: 64, 4-way 2M/4M: 32, 4-way 1G: fractured</td>
<td>4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way</td>
<td>4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way</td>
</tr>
<tr>
<td>L2 Unified TLB</td>
<td>4K: 512, 4-way</td>
<td>4K: 512, 4-way</td>
<td>4K+2M shared: 1024, 8-way</td>
</tr>
</tbody>
</table>

All caches use 64-byte lines
## Buffer Sizes

Extract more parallelism in every generation

<table>
<thead>
<tr>
<th></th>
<th>Nehalem</th>
<th>Sandy Bridge</th>
<th>Haswell</th>
<th>Skylake</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out-of-order Window</td>
<td>128</td>
<td>168</td>
<td>192</td>
<td>224</td>
</tr>
<tr>
<td>In-flight Loads</td>
<td>48</td>
<td>64</td>
<td>72</td>
<td>72</td>
</tr>
<tr>
<td>In-flight Stores</td>
<td>32</td>
<td>36</td>
<td>42</td>
<td>56</td>
</tr>
<tr>
<td>Scheduler Entries</td>
<td>36</td>
<td>54</td>
<td>60</td>
<td>97</td>
</tr>
<tr>
<td>Integer Register File</td>
<td>N/A</td>
<td>160</td>
<td>168</td>
<td>180</td>
</tr>
<tr>
<td>FP Register File</td>
<td>N/A</td>
<td>144</td>
<td>168</td>
<td>168</td>
</tr>
<tr>
<td>Allocation Queue</td>
<td>28/thread</td>
<td>28/thread</td>
<td>56</td>
<td>64/thread</td>
</tr>
</tbody>
</table>
Haswell Core μArch

32KB L1 I-Cache → Pre decode → Instruction Queue → Decoders → μop Cache

Branch Prediction Unit

Allocate/Rename/Retire

Scheduler

Port 0
- ALU
- Shift
- JMP

Port 1
- ALU
- LEA
- MUL

Port 5
- ALU
- LEA

Port 6
- ALU
- Shift
- JMP

Port 4
- Store Data

Port 2
- Load
- Store Address

Port 3
- Load
- Store Address

Port 7
- Store Address

Load Data 2
Load Data 3

Memory Control

256KB ML$

Fill Buffers

32KB L1 D$

From the Optimization Manual

Computer Structure 2015 – Intel® Core™ μArch
Hyper Threading Technology
Thread-Level Parallelism

- Multiprocessor systems have been used for many years
  - There are known techniques to exploit multiprocessors
  - Chip Multi-Processing (CMP): multiple Cores on the same die

- Software trends
  - Applications consist of multiple threads or processes that can be executed in parallel on multiple processors

- Thread-level parallelism (TLP) – threads can be from
  - The same application
  - Different applications running simultaneously
  - Operating system services

- Increasing single thread performance becomes harder
  - And is less and less power efficient
Multi-Threading Schemes

- **Multi-threading**: a single processor executes multiple threads
- **Most applications’ sustained throughput** << the core’s peak IPC
- **Running ≥2 threads increases the utilization of core resources**
  - When one thread is stalled (on a cache miss, branch misprediction, or a long dependency), the other thread gets to use the free resources
  - The RS can find more independent μops
- **Switch-on-event multithreading**
  - Switch threads on long latency events such as last level cache misses
  - Works well for server applications that have many cache misses
  - Does not cover for branch mispredictions and long dependencies
- **Simultaneous multi-threading (SMT)**
  - Multiple threads execute on a single processor simultaneously w/o switching
  - When one thread is stalled / slowed down, other thread gets more resources
  - Makes the most effective use of processor resources
Hyper-Threading Technology

- Hyper-Threading Technology implements SMT

- Two logical processors within one physical core
  - Sharing most execution/cache resources using SMT
  - Look like two processors to the SW (OS and apps)

- Each logical processor executes a software thread
  - Threads execute simultaneously on one physical core

- Each logical processor maintains its own arch. state
  - Complete set of architectural registers
    - General-purpose registers, Control registers, Machine state registers, Debug registers
    - Instruction pointers

- Each logical processor has its own interrupt controller
  - Handles the interrupts sent to the specific logical processor
SMT: A High-level View of the Pipeline

- Each pipe-stage is occupied by one of the threads
- Buffers are either Replicated or partitioned
SMT: Thread Select Points

- Pipeline arbitration points select the thread gets to use a resource in a given cycle
  - If both threads are active and require the resource
    - Use a “ping-pong” scheme to switch between the two threads on a cycle-by-cycle basis
  - If only one thread has work to do and requires the resource
    - Allow that thread to get the full resource bandwidth

- The thread select scheme
  - Achieves fairness between the threads
  - Gives full bandwidth to one thread in case the other thread does not require the given resource
SMT: Thread Select Points

- The key arbitration points in the pipeline select in a given cycle which thread gets to:
  - Read the I$ to Fetch instruction
  - Use the Decodes for decoding instructions into μops
  - Allocate μops to the ROB, store/load buffer, and the RS
  - Retire μops and reclaim their resources

- The RS is not a thread arbitration point:
  - The RS schedules μops to the execution units based on readiness and based on age (allocation time)
  - Regardless of which thread they belong to
SMT Principals

• When one thread is stalled, the other thread can continue to make progress
  - Independent progress ensured by either
    ➢ Partitioning buffering queues and limiting the number of entries each thread can use
    ➢ Duplicating buffering queues

• A single active thread running on a processor with HT runs at the same speed as without HT
  - Partitioned resources are recombined when only one thread is active

Based on ITJ Vol. 14, Issue 3, 2010 Intel® Core™ next gen uarch
Physical Resource Sharing Schemes

- **Replicated Resources:** e.g. CR3
  - Each thread has its own resource

- **Partitioned Resources:** e.g. µop-Cache
  - Resource is partitioned in MT mode, and combined in ST mode

- **Competitively-shared resource:** e.g. RS entries
  - Both threads compete for the entire resource

- **HT unaware resources:** e.g. execution unit
  - Both threads use the resource

- **Alternating Resources**
  - Alternate between active threads
  - If one thread idle
    - The active thread uses the resource continuously
Front End Resource Sharing

- The Instruction Pointer is replicated
- The small page iTLB is partitioned
- Branch prediction resources are mostly shared
- The I$ is thread unaware (thus competitively-shared)
- The pre-decode logic and the instruction decoder logic are used by one thread at a time
- The μop-Cache is partitioned
- The μop Queue is partitioned
Back End Resource Sharing

- **Out-Of-Order Execution**
  - Register renaming tables are replicated
  - The reservation stations are competitively-shared
  - Execution units are HT unaware
  - The re-order buffers are partitioned
  - Retirement logic alternates between threads

- **Memory**
  - The Load buffers and store buffers are partitioned
  - The DTLB and STLB are competitively-shared
  - The cache hierarchy and fill buffers are competitively-shared
Single-task And Multi-task Modes

- **MT-mode (Multi-task mode)**
  - Two active threads, with some resources partitioned as described earlier

- **ST-mode (Single-task mode)**
  - ST0 / ST1 – only thread 0 / 1 is active
  - Resources that are partitioned in MT-mode are re-combined to give the single active logical processor use of all of the resources

- **Moving the processor from between modes**

![Diagram showing the transition between modes](image)
Thread Optimization

The OS should implement two optimizations:

- **Use HALT if only one logical processor is active**
  - Allows the processor to transition to either the ST0 or ST1 mode
  - Otherwise the OS would execute on the idle logical processor a sequence of instructions that repeatedly checks for work to do
  - This so-called “idle loop” can consume significant execution resources that could otherwise be used by the other active logical processor

- **On a multi-processor system**
  - OS views logical processors similar to physical processors
    - But can still differentiate and prefer to schedule a thread on a new physical processor rather than on a 2nd logical processors in the same physical processor
  - Schedule threads to logical processors on different physical processors before scheduling multiple threads to the same physical processor
  - Allows SW threads to use different physical resources when possible