3. Explicit concurrency

VLIW and embedded processors

Advances in Computer Architecture

Summary of explicit concurrency (1)

- **Power dissipation** is one of the major issues in obtaining performance
  - particularly true with constraints on embedded systems
- **Concurrency** can be used to **reduce power** and **get performance**
  - reduce frequency which in turn allows a voltage reduction
  - does not apply to pipelined concurrency where more concurrency increases power due to shared structures

OoO not scalable – what else?

- Out-of-order issue is not scalable
  - register file and issue logic scale as ILP^3 and ILP^2 respectively
- A number of approaches have been followed to increase the utilisation of on-chip concurrency
- These include:
  - VLIW processors
  - Speculative VLIW processors - Intel's IA64 - EPIC
  - multi-core and multi-threaded processors (later on)

Summary of explicit concurrency (2)

- **VLIW** is the simplest form of explicit concurrency
  it **reduces hardware complexity, hence power requirements**
  - drawbacks are **code compatibility** and intolerance to **cache misses**
- **EPIC** attempts to resolve these problems in VLIW
  - this **adds complexity** and **requires large caches** to get good performance
- Multi-cores are scalable but **need new programming models**
  - the big question is - can we design general purpose multi-cores?

Very Large Instruction Word (VLIW) processors

Explicit concurrency in hardware, compiler in charge of scheduling

Overview

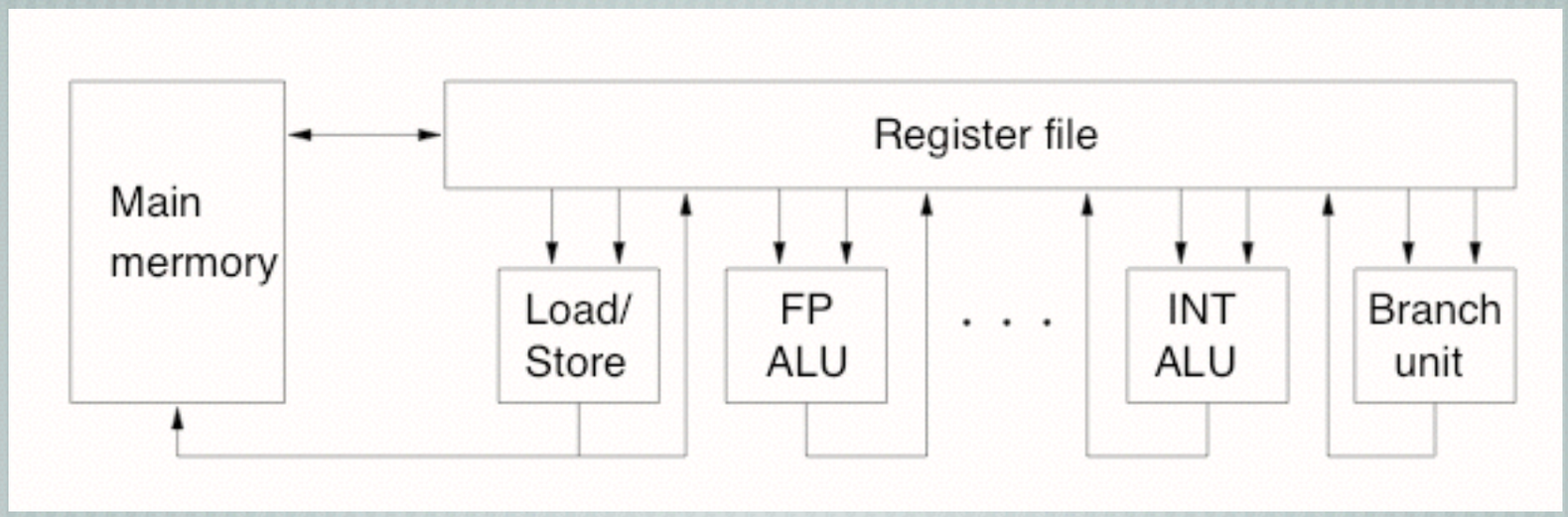
- **VLIW** was introduced as parallel micro-coded processors in the 1970s
  - it was originally called **horizontal microprogramming**
  - the principle was to explicitly control the operation of multiple functional units on a cycle-by-cycle manner
  - today VLIW puts several RISC-like operations into one wide instruction which is implemented with very short and efficient pipelines
- This approach puts **the scheduling problem in the domain of the compiler**, which needs to know everything about the architecture
  - detailed timing is required as instructions are scheduled for issue every cycle by the compiler - new processor, new code needed

Scalable instruction issue

- **VLIW: no logic required to schedule operations and find dependencies**
  - this scheduling is performed in the compiler or assembly programmer
  - concurrency is analyzed by the compiler;
    independent program operations are grouped into a single VLIW instruction
  - the compiler manages the scheduling of data dependencies
- In contrast to OoOE, VLIW instruction issue is scalable;
  there is no adverse growth in complexity due to wider issue width
  - there are still scaling problems with the register file but the register file can be partitioned with explicitly communication of data between partitions

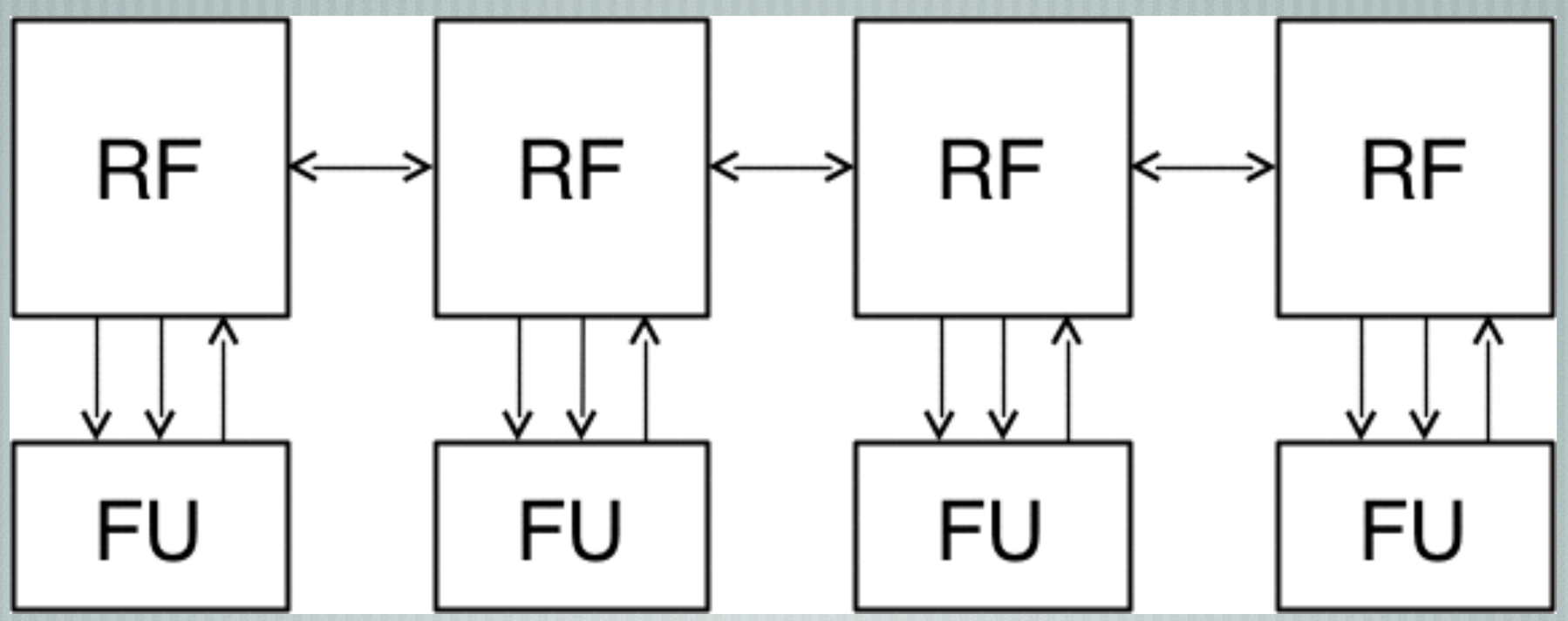
Register file scalability

- The ideal VLIW architecture is shown below
  - it has a number of functional units, all accessing a single register file
  - this is NOT scalable due to the scaling of the register file with number of access ports
- Alternative solutions divide units into clusters with their own register file
- This complicates the scheduling as compiler must now schedule inter-cluster data movements



Distributed/Shared Register files

- Use explicit transfers between register files (RF) associated with one or more functional units (FU)



VLIW Advantages & Drawbacks

Advantages:

- Simple pipelines with low latency
- Greater levels of concurrency possible
  - compilers can expose more concurrency (in theory) - no hardware limits in the search window
  - more area on chip available for functional units
- Good performance in some applications
  - multimedia
  - other regular applications

Drawbacks:

- No binary code compatibility
- Register file still not scalable
  - must cluster function units
  - more difficult to schedule
- Branch prediction usually performed statically
- Code is less compact due to padding
- Cannot schedule for unknown delays in executing instructions
  - a cache miss means stalling all operations

VLIW vs. binary compatibility

- A major disadvantage of VLIW is the loss of binary code compatibility
  - every increase in concurrency means a new instruction format and the source code must be recompiled
  - this is not an issue in embedded microprocessors but is an issue in commodity microprocessors
- Transmeta Crusoe/Efficeon uses VLIW to implement an Intel X86 "compatible" processor - using **dynamic code translation** "Code Morphing" however this requires huge translation caches

VLIW vs. code density

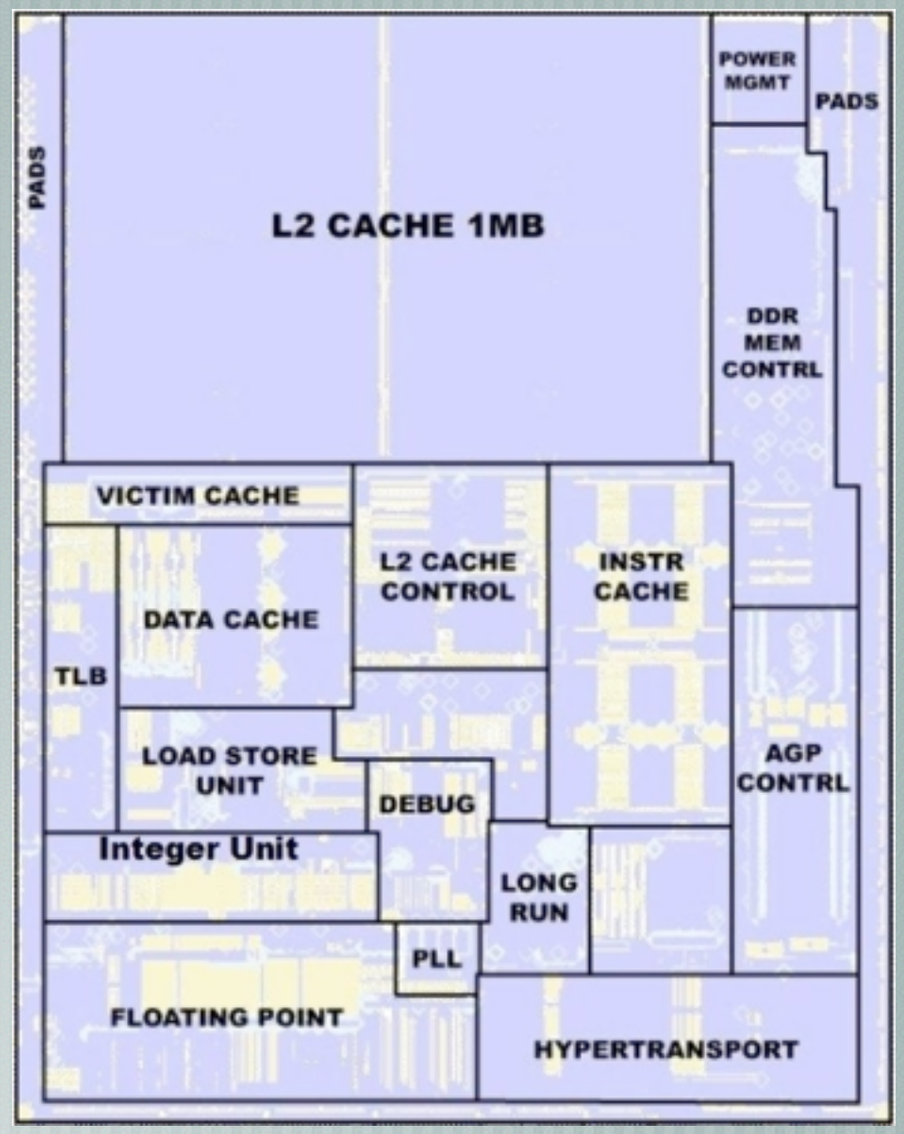
- If the VLIW compiler can not find instructions to schedule, it must **pad the instruction words with no-ops**
  - for long data dependencies this may mean many cycles with few or no active instructions
  - this increases code size relative to a sequential instruction stream
  - it is a big problem in the application domain where VLIW is used - embedded systems - here small code memories are typical due to power and size constraints
- The solution is to **compress the code**
  - no-ops give long runs of zeros in code which can easily be compressed
  - Decompression occurs during either
    I-cache fill (larger caches needed), or fetch (impacts the critical path)

VLIW vs. the memory hierarchy

- VLIW **works best with no memory hierarchy**
  - eliminates variable access access times in memory, with fixed latencies accesses scheduled statically by the compiler
  - small embedded processors tend to use small cache-less SRAM memories, this is why VLIW is still common in these systems
- Compilers for cache-based VLIW will normally schedule for a cache hit on each load instruction...
- A cache miss would then...
  - stall all instruction issue until data was found in a lower level, or
  - cause an exception so that another processes can continue
- Either way it gives a high overhead which becomes worse as issue width increases, i.e. more stalls

Example: TM8800 Efficeon (2004)

- 1GHz 256-bit VLIW engine (8x 32 bit RISC instructions)
  - 128KByte L1 I-cache, 64KByte L1 D-cache
  - 1MByte L2 cache on chip
  - 32MByte translation cache in main memory
- Fabricated in the 90nm Fujitsu process
- Code compatible with X86 through "code-morphing" software layer
  - dynamically translates X86 binary code into VLIW instructions: this involves scheduling instructions in the VLIW instruction words
  - similar to compiling Java with JIT (just in time) compilers except in the TM Efficeon there is some hardware support
  - translate and cache code as it is executed, repetitive code sequences get reused and give good efficiency and performance



Low-power – Transmeta runs cold

- Uses low frequencies & high levels of concurrency
- Uses dynamic frequency-voltage control
- Claimed up to 10 hours run time on laptop battery

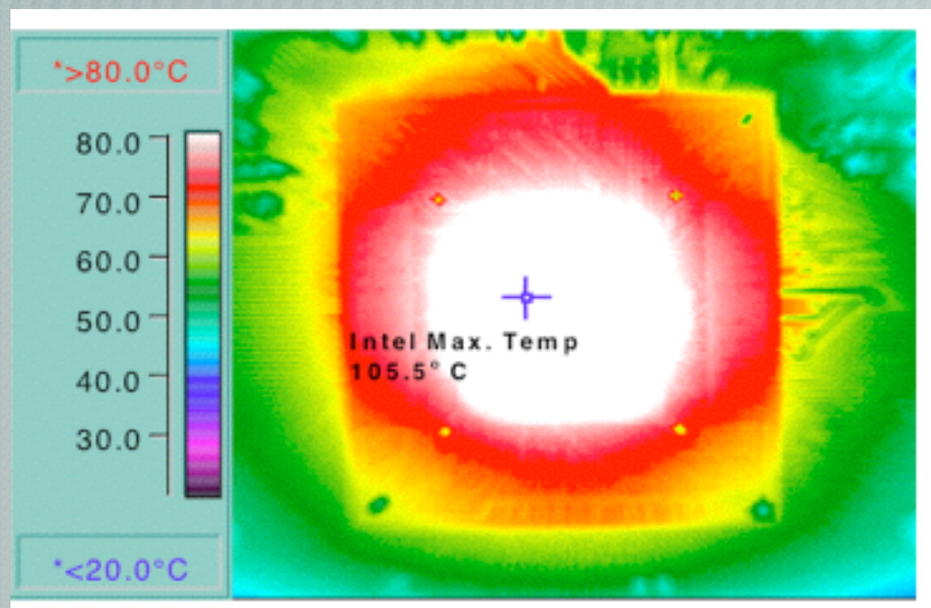


Figure 3. A Pentium III processor plays a DVD at 105° C (221° F).

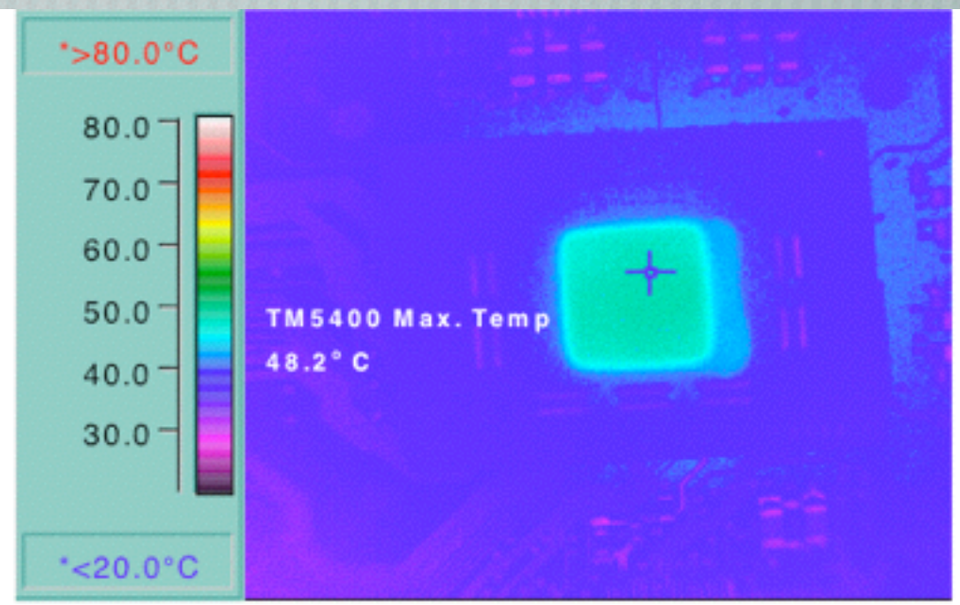
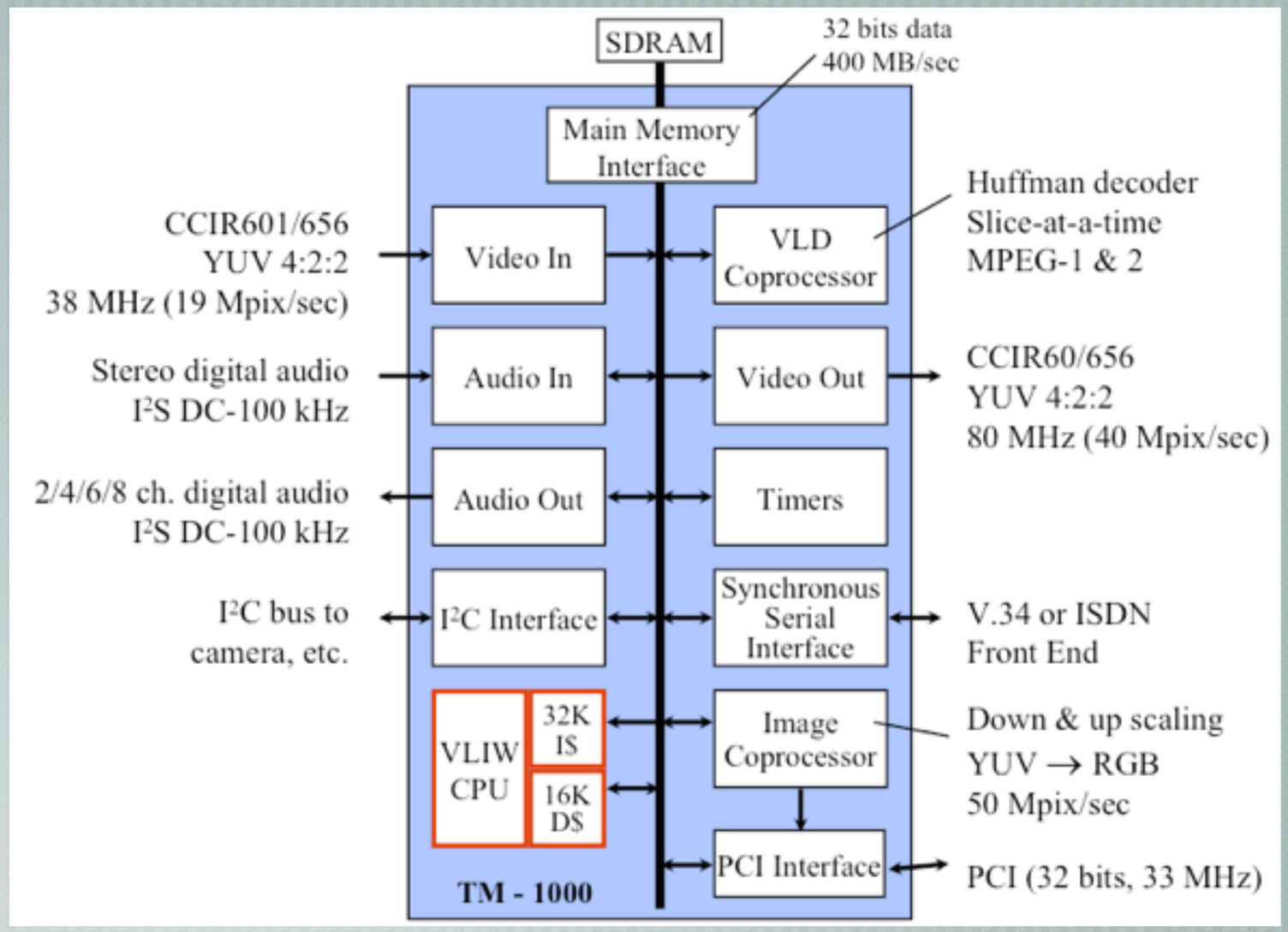


Figure 4. A Crusoe processor model TM5400 plays a DVD at 48° C (118° F).

Philips TriMedia TM 1000 (late 90's)

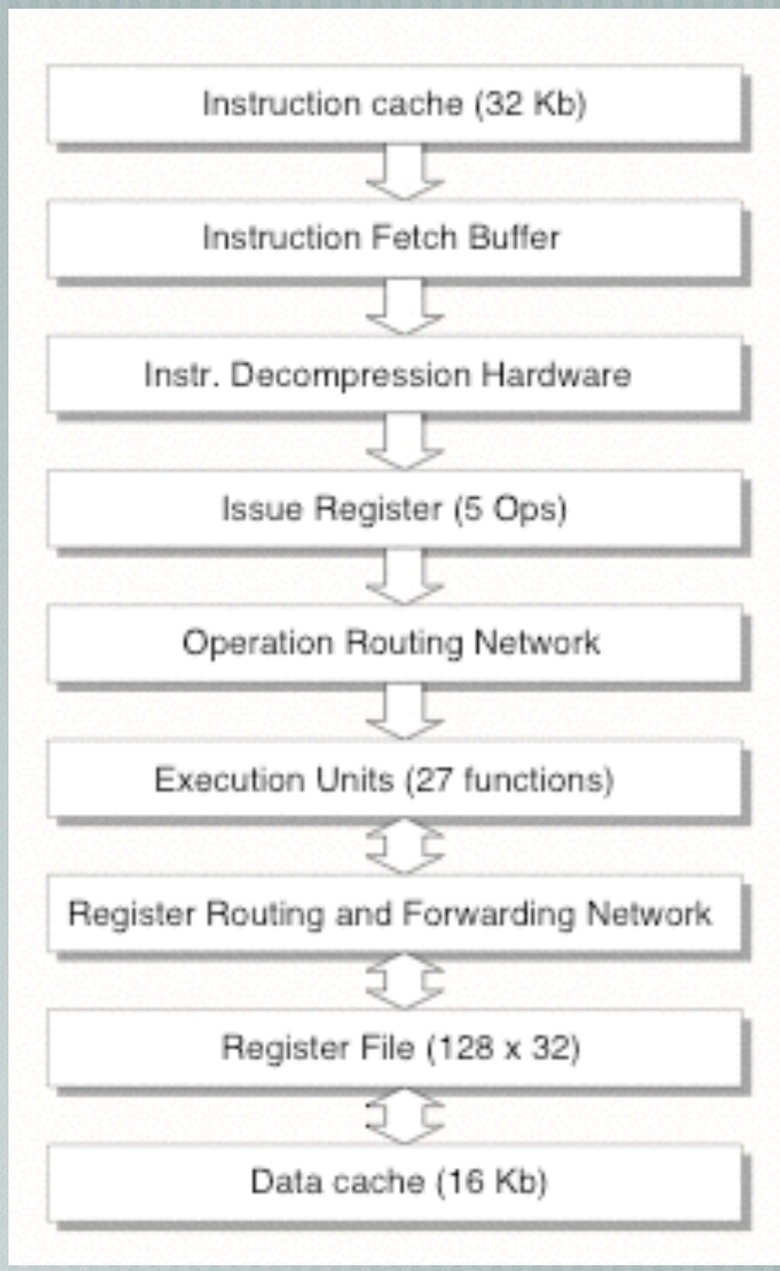
- 32-bit VLIW media processor
- Five-op VLIW instruction (2 load/stores)
- Predicated instructions
- SIMD extensions
- Co-processors for common operations



RiP: TriMedia died in 2010

TM 1000 Pipeline

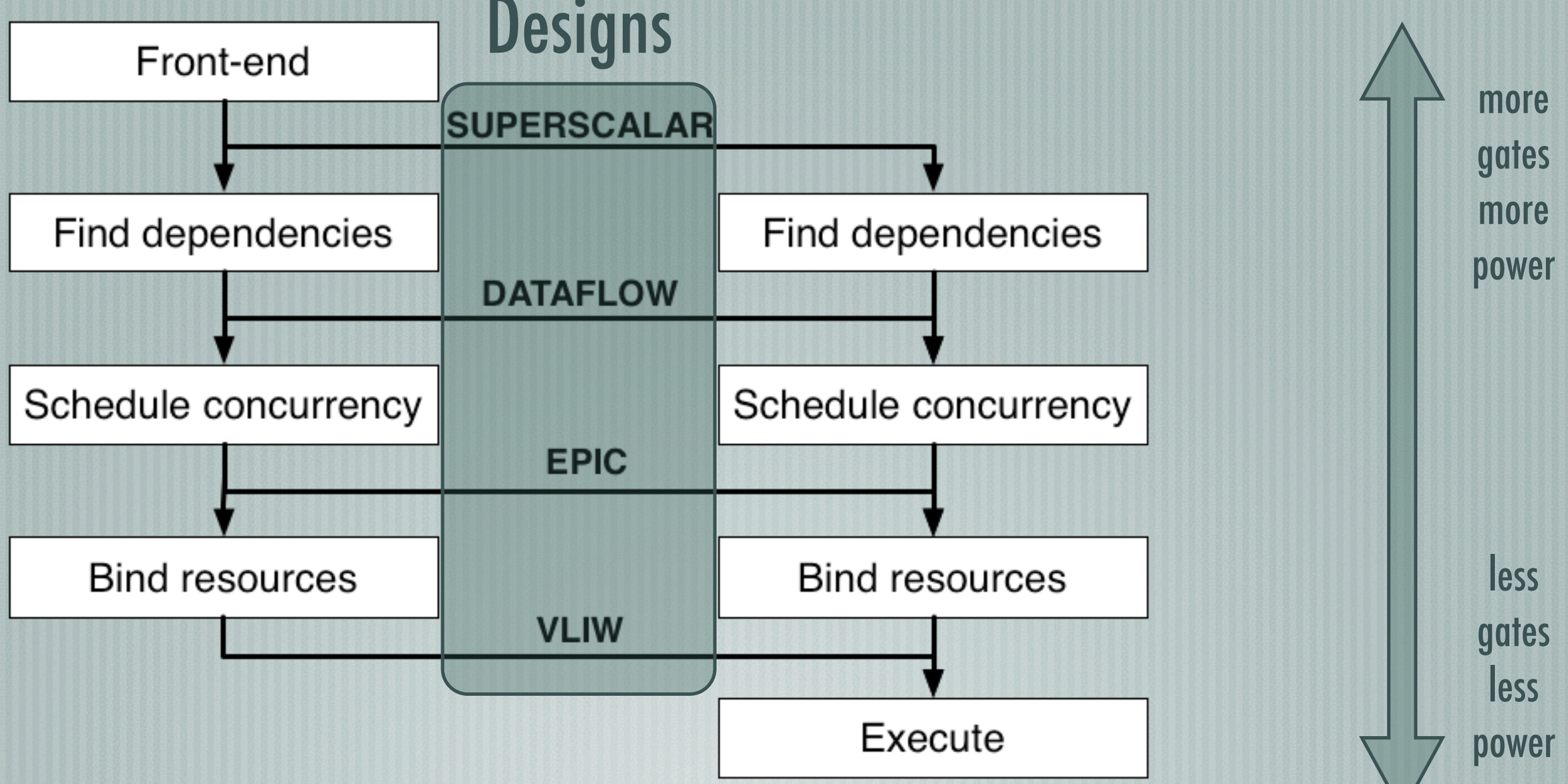
- 220bit = 5 x 44-bit instruction words issued to 27 execution units
- 128 register file (15 read and 5 write ports) requires complex register forwarding network across 32 files
- 32 Kbyte I-cache 8-way set associative 64 byte line
- 16KByte 8-way set associative non-blocking D-cache
  - 8 banks, (pseudo) dual-ported
  - Streamed, critical-word-first, fetching
  - Programmer controlled prefetching + allocation



HP/Intel IA 64 - Itanium 2

- Problems with conventional VLIW:
  - cache misses break compile time schedule - not known at compile time so cannot be factored into the schedules
  - control hazards
  - code compatibility across different generations of issue width
- Only the latter has a real solution, yet impractical (recompile, or JIT translation)
- **IA 64 was designed to solve all three problems**
  - a cross between speculative RISC and VLIW called **EPIC**
  - see http://www.cs.clemson.edu/~mark/epic.html for details and historical background

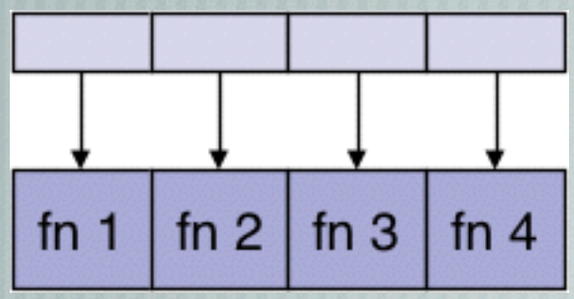
Spectrum - static vs dynamic



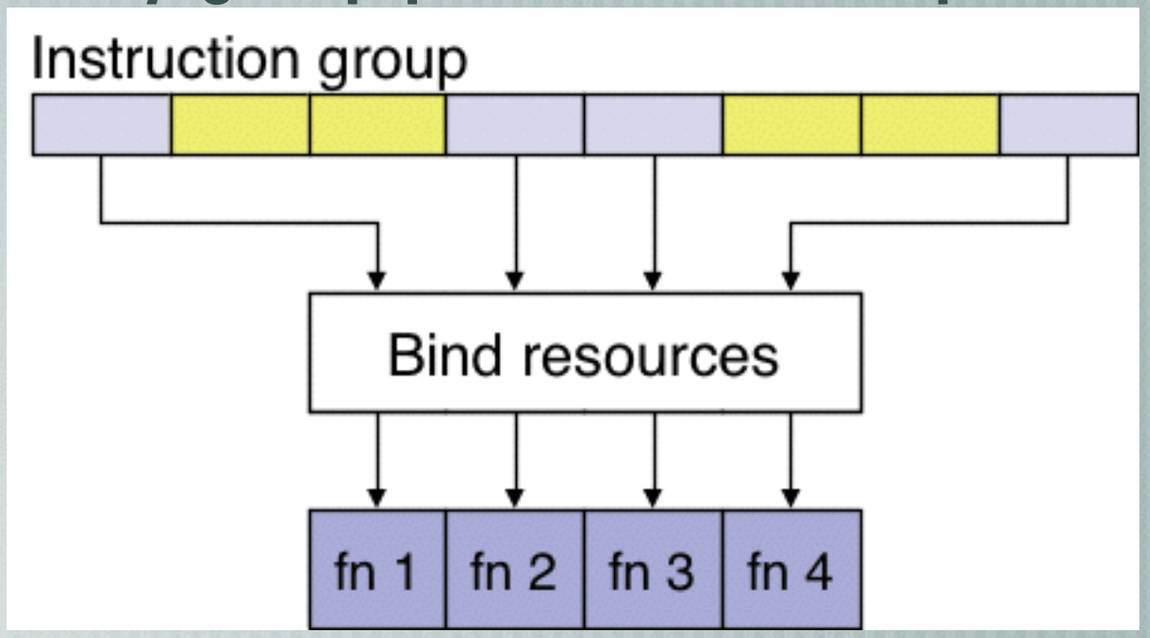
EPIC's Instruction packets

- VLIW compilers bind concurrency to resources via their position in the wide instruction word
- EPIC clusters groups of instructions which are free to execute together but may be executed sequentially; many group patterns can be specified

VLIW



EPIC



EPIC's instruction bundles

- Instructions (syllables) are 41 bits wide
- 3 syllables are grouped into 128 bit bundles including a 5-bit template code (3*41+5=128)
- Template specifies information about the types of instructions contained in the bundle
- An instruction group is then a set of bundles that may execute in parallel

Table 3-9. Template Field Encoding and Instruction Slot Mapping

Template	Slot 0	Slot 1	Slot 2
00	M-unit	I-unit	I-unit
01	M-unit	I-unit	I-unit
02	M-unit	I-unit	I-unit
03	M-unit	I-unit	I-unit
04	M-unit	L-unit	X-unit
05	M-unit	L-unit	X-unit
06			
07			
08	M-unit	M-unit	I-unit
09	M-unit	M-unit	I-unit
0A	M-unit	M-unit	I-unit
0B	M-unit	M-unit	I-unit
0C	M-unit	F-unit	I-unit
0D	M-unit	F-unit	I-unit
0E	M-unit	M-unit	F-unit
0F	M-unit	M-unit	F-unit
10	M-unit	I-unit	B-unit
11	M-unit	I-unit	B-unit
12	M-unit	B-unit	B-unit
13	M-unit	B-unit	B-unit
14			
15			
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit
1A			
1B			
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit
1E			
1F			

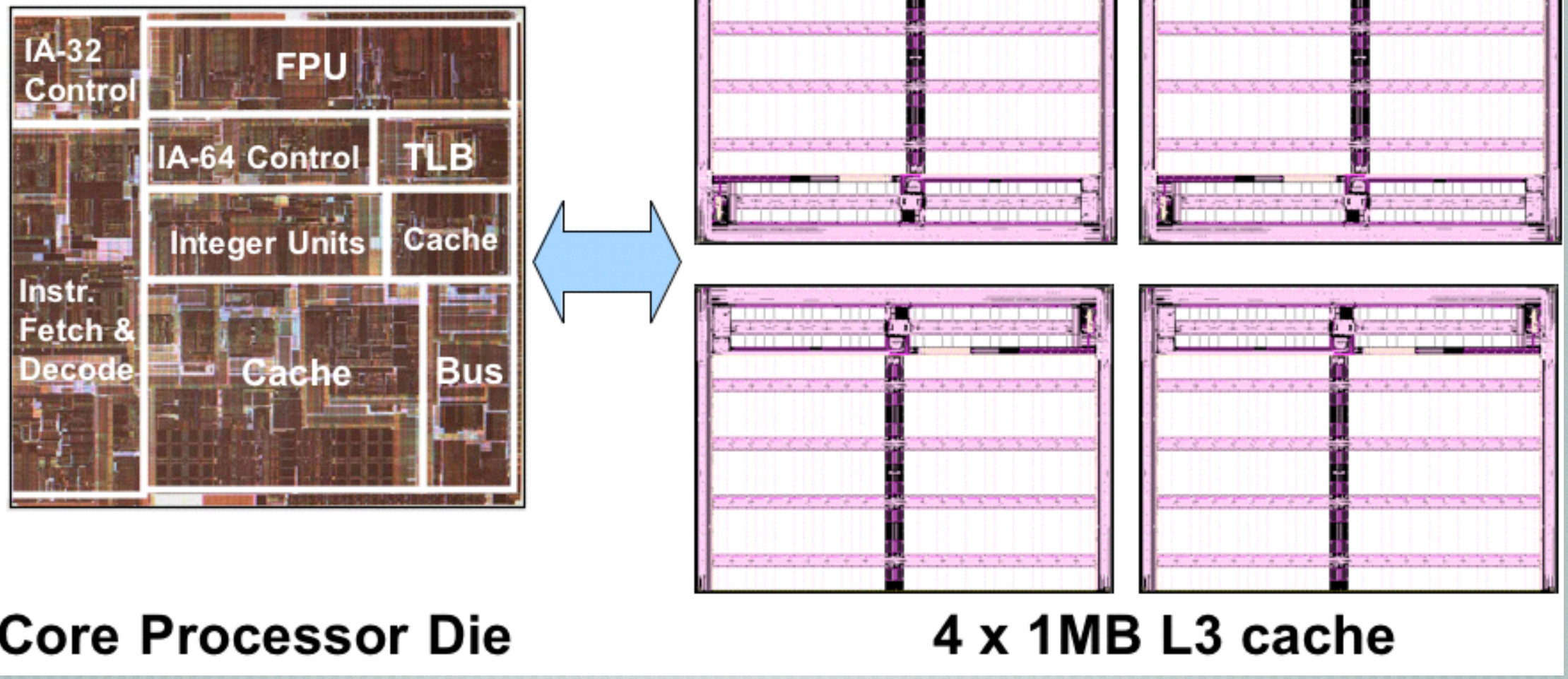
IA-64 Memory accesses

- **Prefetch is typically non-binding**
  - i.e. can be bumped out of cache by conflict
- IA 64 uses **speculative or early loads**
  - follow by check instruction to ensure the load has completed
- IA 64 also provides **control of caching**
  - can specify not to allocate in cache if it is known it will be used only once - e.g. streaming
  - can specify an appropriate cache level

IA-64 Decoupled loads

- Issuing loads as early as possible solves the long latency problem in memory, but ...
  - There may be **writes a compiler can not detect** to the same address, which limit how high a load can be hoisted
- IA-64 uses the LD.A instruction to **speculatively load from memory** - stores the load address into a special buffer called the Advanced Load Address Table (ALAT).
  - Subsequent stores to memory are checked against this
  - Any match aborts the load or discards data if completed
- Need to validate loads with a CHK.A instruction first before the data can safely be used - if not, branch and retry: expensive like mis-predictions in superscalar

Itanium – Merced silicon



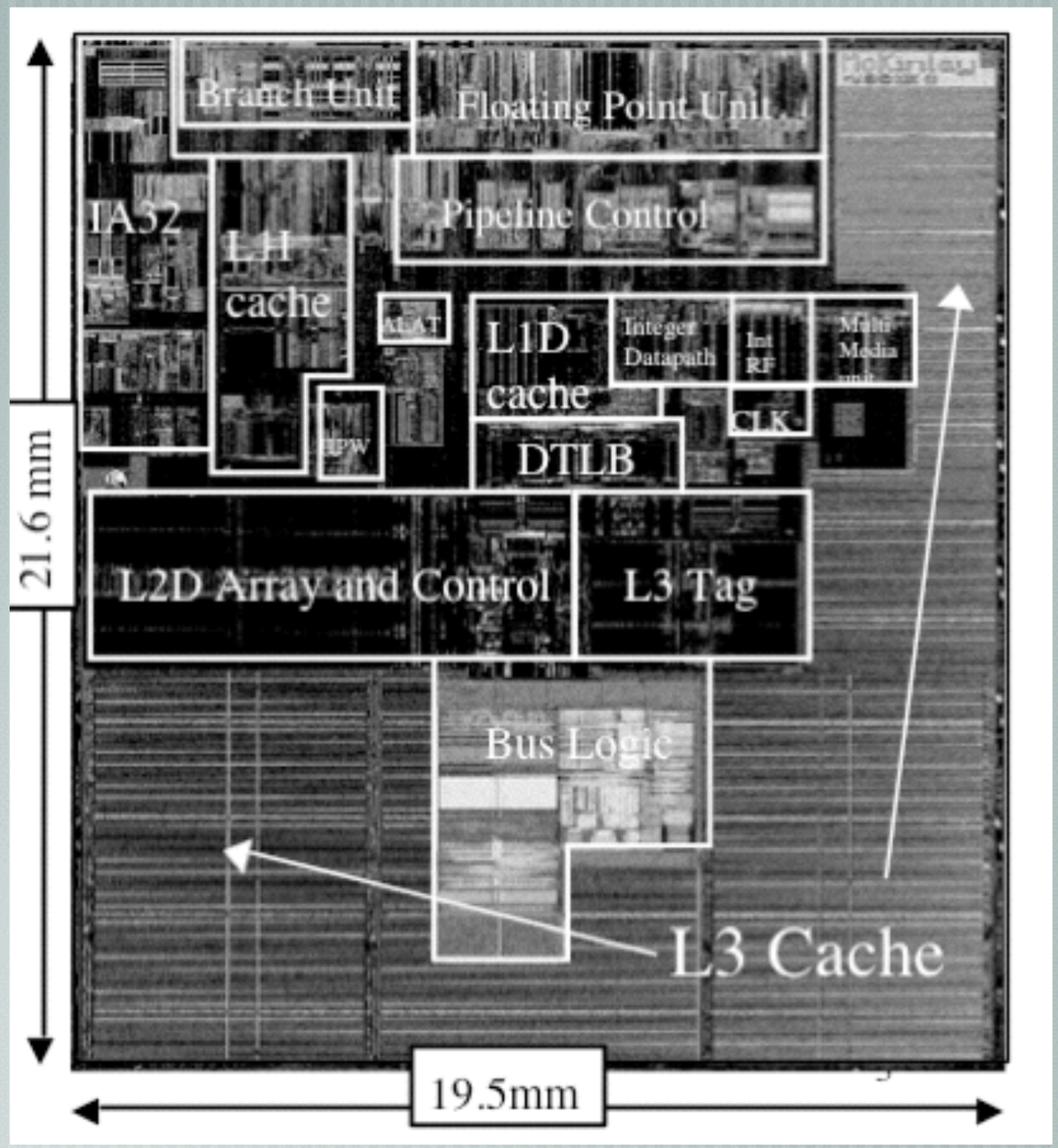
© Intel, HotChips 2000

Merced parameters

Frequency	800 MHz
Transistor Count	25.4M CPU; 295M L3
Process	0.18u CMOS, 6 metal layer
Package	Organic Land Grid Array
Machine Width	6 insts/clock (4 ALU/MM, 2 Ld/St, 2 FP, 3 Br)
Registers	14 ported 128 GR & 128 FR; 64 Predicates
Speculation (advanced loads)	32 entry ALAT, Exception Deferral
Branch Prediction	Multilevel 4-stage Prediction Hierarchy
FP Compute Bandwidth	3.2 GFlops (DP/EP); 6.4 GFlops (SP)
Memory -> FP Bandwidth	4 DP (8 SP) operands/clock
Virtual Memory Support	64 entry ITLB, 32/96 2-level DTLB, VHPT
L2/L1 Cache	Dual ported 96K Unified & 16KD; 16KI
L2/L1 Latency	6 / 2 clocks
L3 Cache	4MB, 4-way s.a., BW of 12.8 GB/sec;
System Bus	2.1 GB/sec; 4-way Glueless MP Scalable to large (512+ proc) systems

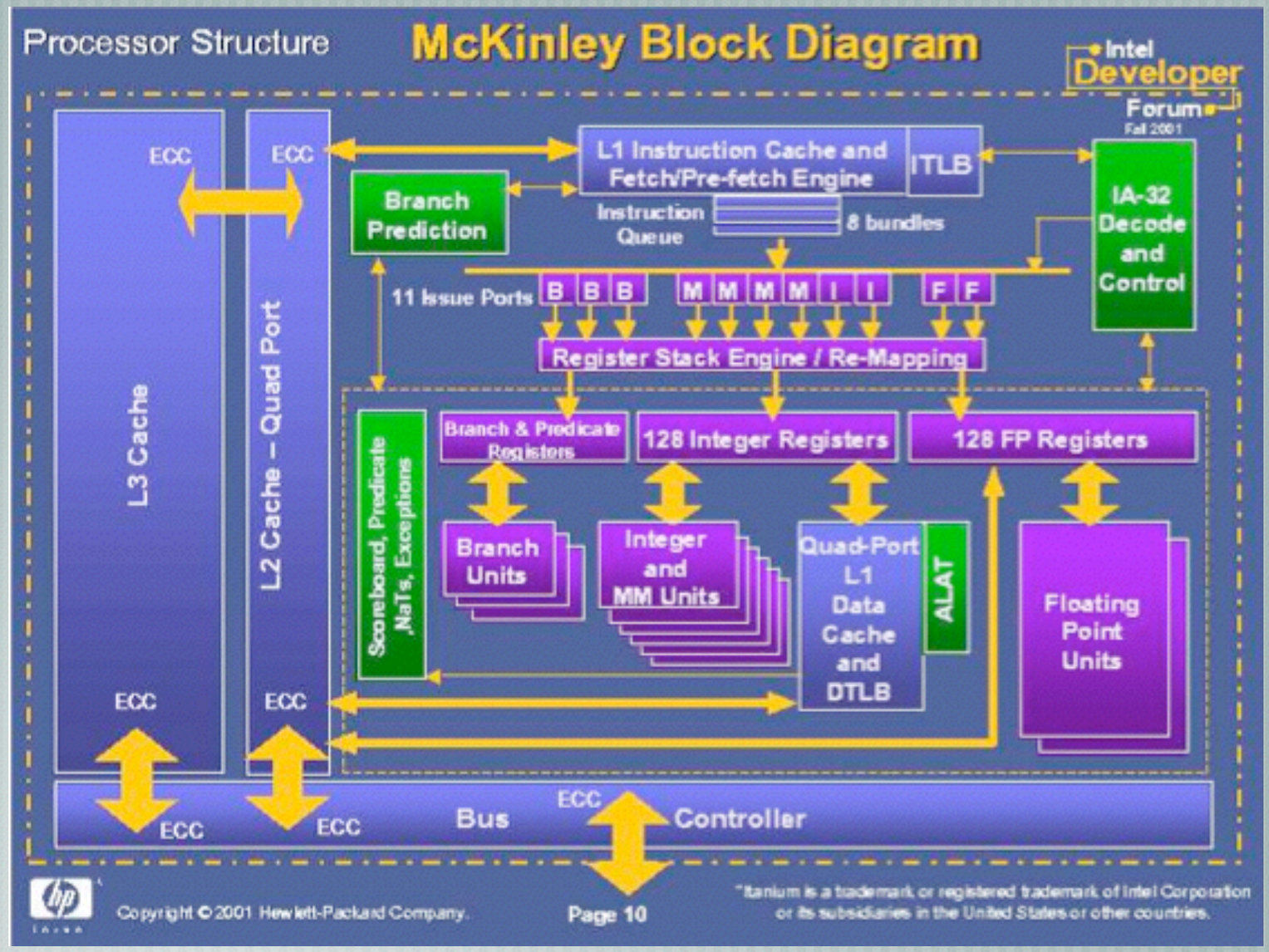
Itanium 2 – McKinley silicon

- 0.18 µm process
- 1GHz core clock
- 200MHz system bus
- Core area 260mm^2



Itanium 2 - McKinley function

- Executes legacy code X86 by translating it to IA64 operatons
- Contains L1, L2 and L3 cache all on chip



Itanium 2 characteristics

- 8-stage in-order pipeline
- Uses **predication** and **branch prediction**
  - 2-level BHT and BTB for branch prediction
  - target address registers for compiler hints
  - loop counter registers
- Up to 3 branches can be issued in parallel
- **6-way instruction issue** from instruction bundles
- Support for loop unrolling using rotating register windows

A power intermezzo

Power dissipation

- Power can be described by the following equation

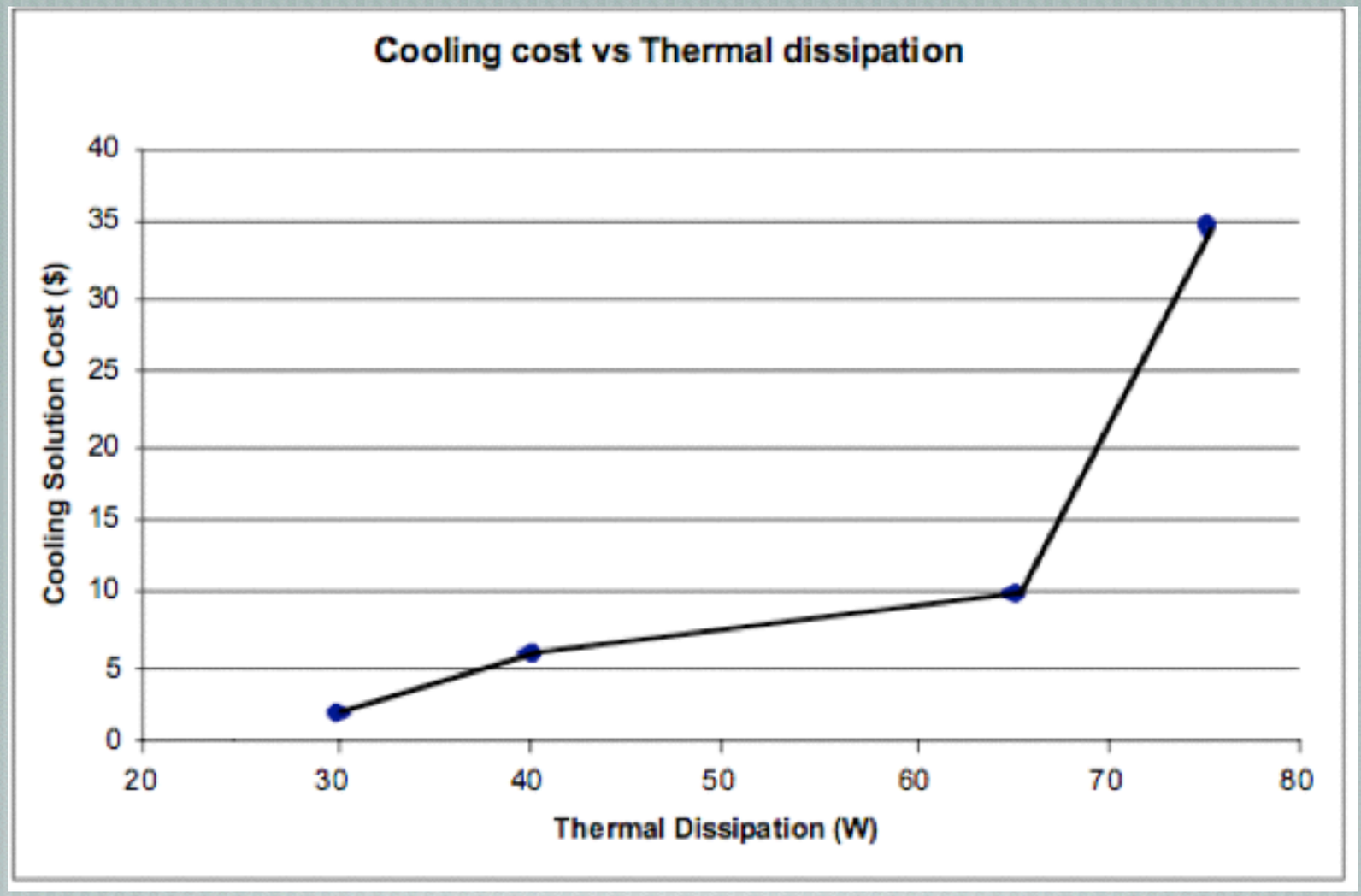
$$P = a \cdot C \cdot V_{dd}^2 \cdot f + a \cdot t \cdot V_{dd} \cdot I_{short} \cdot f + V_{dd} \cdot I_{leak}$$

- 1st term dominates (3rd will soon overtake it)
  - C is capacitance being charged by gates, V_{dd} is the suply voltage, f is frequency, a is gate activity (rate of gates that switch)
- 2nd term is power dissipated when the power and ground rails are shorted for a brief period during switching
  - I_{short} is the short circuit current, t the transition time
- 3rd term is power dissipated in leakage across gates
  - I_{leak} is the leakage current, V_{dd} the input voltage

Power-related constraints

- Problems caused by power dissipation (heat):
  - thermal runaway - the leakage currents of transistors increase with temperature
  - electro-migration diffusion – metal migration with current increases with temperature (fuse effect)
  - electrical parameters shift - CMOS gates switch faster when cold (cf “to over-clock get some liquid nitrogen!”)
  - silicon interconnections fatigue (expand and contract with temp. swings)
  - package related failure
- Power dissipation constraints (e.g. ≈100W per package) provide an upper limit on power usage, and thus component frequency and input voltage

Cooling costs



From: S.H. Gunther, Managing the impact of increasing microprocessor power consumption, Intel Technology Journal Q1, 2001

Reducing power

- Reducing V_{dd} effectively reduces power consumption (quadratic relationship!)
  - **But**, reducing V_{dd} also limits the maximum frequency
    - $f_{max} \propto (V_{dd} - V_{threshold})^2/V_{dd} \rightarrow f_{max}$ roughly linear to V_{dd}
  - Lessen the effect by reducing $V_{threshold}$
  - Unfortunately, this increases leakage current
    - $I_{leak} \propto e^{-V_{threshold}/35mV}$
- Many processors use Dynamic Voltage / Frequency Scaling to reduce power (controlled by OS)

Reducing power - architecture level

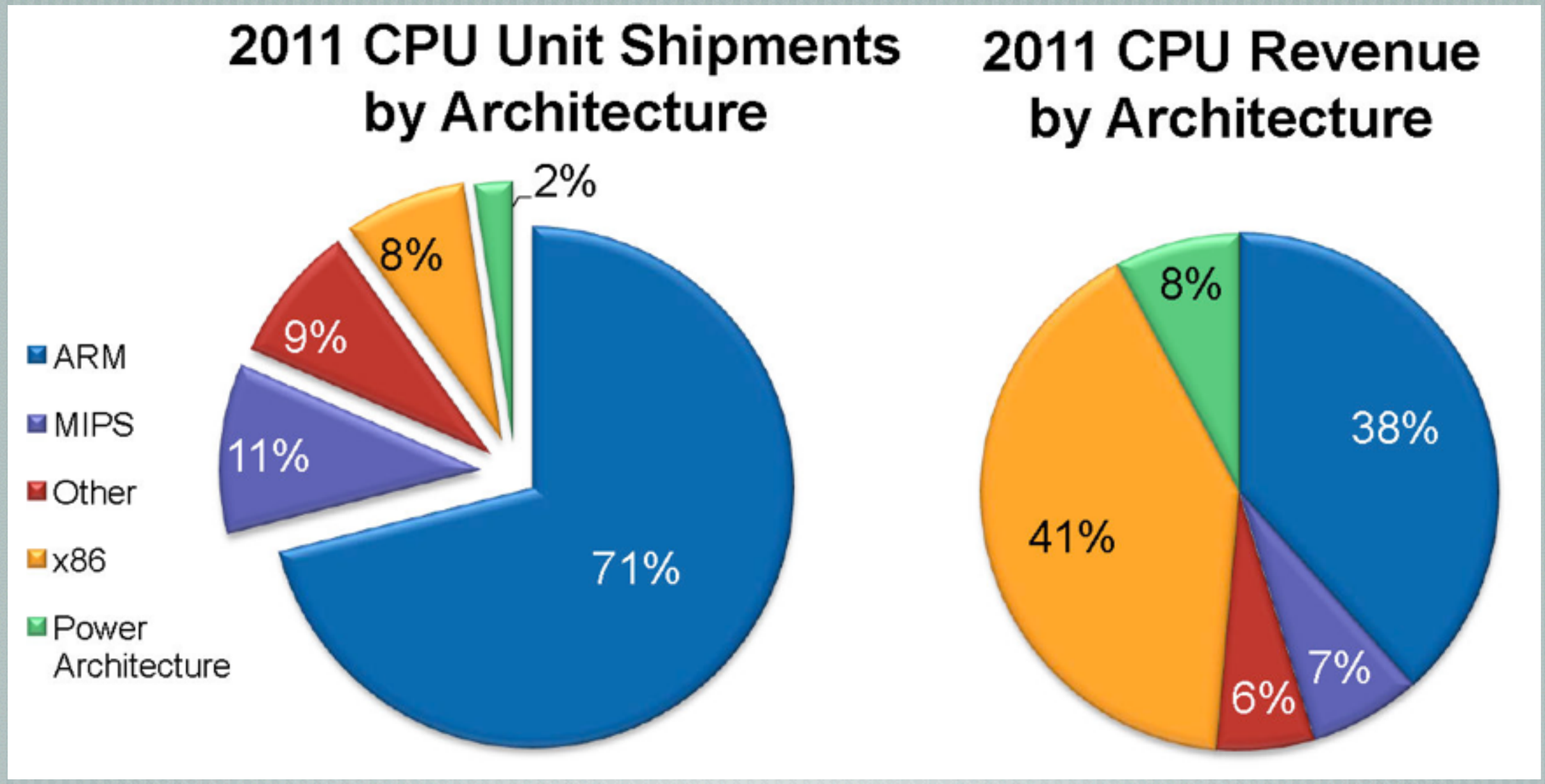
- Avoid speculation and issue instructions conservatively
  - any misspredicted speculation requires power dissipation for no tangible results
- Organization of memory: multiple smaller banks
- Avoid unnecessary operations
  - e.g. reading registers where data is bypassed
- Reduce number of swings on data busses
  - e.g. use Gray codes to exploit locality
- Only send bits that change - form of data compression
- And, of course, **exploit parallelism** (i.e., reduce frequency)!

Reducing power - logic/circuit level

- Clock gating
  - avoid transitions in logic where no activity is taking place; can also save on clock distribution to those areas
- Double clocking with half rate clocks
  - use both edges of clock pulse to reduce power dissipation in clock distribution - does not reduce logic transitions
- Asynchronous logic design
  - only dissipate power on gate transitions that are required

Embedded processors

Embedded processors



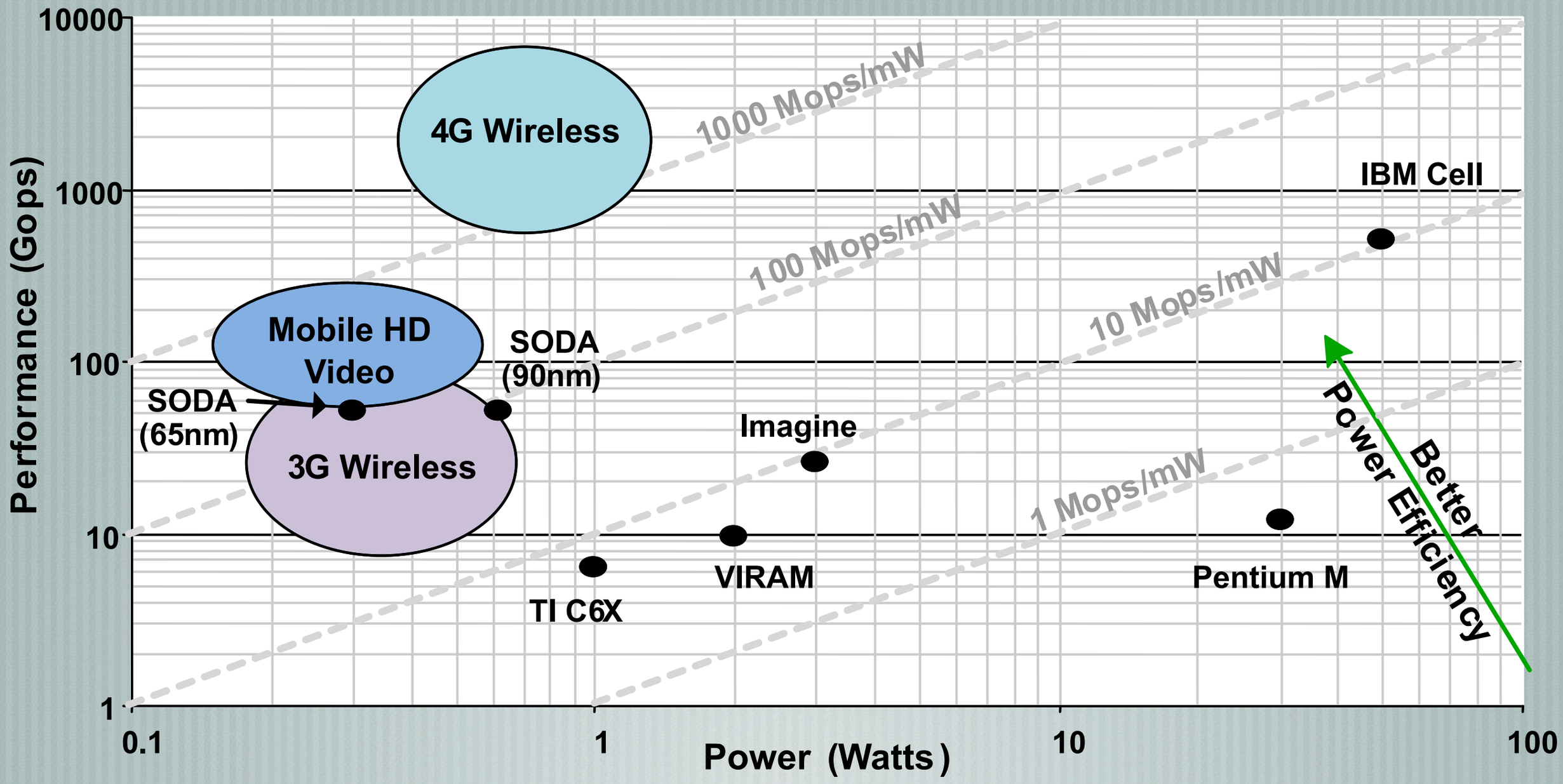
Endless applications

- To name some obvious ones
  - Smart phones
  - Car engine control systems
  - Microwave cookers
  - Washing machines
  - Surveillance cameras
  - Smart military ordinance
- The list goes on...

Embedded system characteristics

- **Real-time constraints**
  - continuous data streams; must process samples as received
- **Low cost** - e.g. cheap commodity items
- **Low power** - battery operated devices, devices without active cooling (e.g., TV) etc.
- **Small memory footprint** - minimal memory available
- **Very reliable** - think automotive/aerospace applications
- Often have a **short time to market**

Computational efficiency



Woh e.a., ISCA 2009

Embedded processors

- General purpose "processor cores"
  - E.g. ARM, MIPS, SPARC, PPC, M68000
- Digital Signal Processors (DSPs)
  - E.g. Texas Instruments, Qualcomm
- Media processors
  - E.g. Philips Trimedia, PS2 Emotion engine, IBM Cell

Embedded processors

- Embedded processors typically are
  - in-order execution engines (for non-performance-critical workloads)
  - VLIW (for heterogeneous workloads)
  - SIMD processors (for regular workloads)
- Often use low-precision fixed point arithmetic using integer ops (instead of power-hungry FP)

Embedded processors

- **Harvard architecture** often used in embedded processors
  - i.e. separate data and instruction memory
  - L1 caches do this, but it was pioneered in pre-cache designs and is still used in cacheless designs (e.g., code in ROM, data in RAM)
- Embedded processors **sometimes avoid caches completely**
  - instead use fast SRAM memories with multiple banks
- **Combined instructions** (e.g., multiply-accumulate) to minimize memory and/or register references
- Embedded processors often support **special addressing modes**, for example:
  - modulo addressing for cyclic buffers
  - bit-reversed addressing for fast-Fourier transforms (FFT)

Embedded system architectures

- Many embedded systems have a **heterogeneous** system architecture that is increasingly integrated on a single chip: **(Multi-Processor) System-on-Chip**
  - GP processor cores, DSPs, Application-Specific Instruction-set Processors (ASIPs), Application-Specific Integrated Circuit (ASICs), etc.
- **Design space is large**, many processors and processor types, choice of software or hardware implementations
  - need to trade off power, performance, cost, reliability, flexibility, etc.
  - but these often are **conflicting objectives**

Embedded systems design

If time permits

Processor design roadmaps across market segments