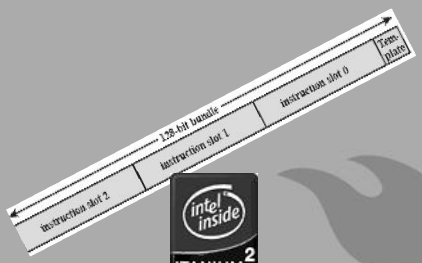
HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

Lecture 11

Itanium
(old name: IA-64)



Stallings: Ch 15

- IA-64 overview
- Predication, speculation
- software pipelining
- Itanium 2

Intel Multi-core and STI Cell

EPIC (Explicit Parallel Instruction Computing)

❶ Parallelism explicit in machine instructions, not hidden inside the hardware (processor)
- New semantics on machine instructions
- **Compiler** solves dependencies and decides the parallel execution issues, processor just trusts it

❷ VLIW (Very Long Instruction Word)
- Handle instructions in bundles (*nippu*)

❸ Branch predication, control speculation
- Speculative execution of both (all) branch targets

❹ Spekulative loading of data

Linuksen kommentti IA-64:sta (2005): http://www.realworldtech.com/forums/index.cfm?action=detail&id=60298&threadid=60123&roomid=2

Itanium vs. Superscalar

(Sta06 Table 15.1)

Superscalar	IA-64
RISC-like instructions, one per word	RISC-like instructions bundled into groups of three
Multiple parallel execution units	Multiple parallel execution units
Reorders and optimizes instruction stream at run time	Reorders and optimizes instruction stream at compile time
Branch prediction with speculative execution of one path	Speculative execution along both paths of a branch
Loads data from memory only when needed, and tries to find the data in the caches first	Speculatively loads data before its needed, and still tries to find data in the caches first

- Itanium new design and architecture
  - Not backward compatible
- HP and Intel co-operated

More transistors per chip

pipelining + RISC => superscalar => more and =>more parallelism and speed

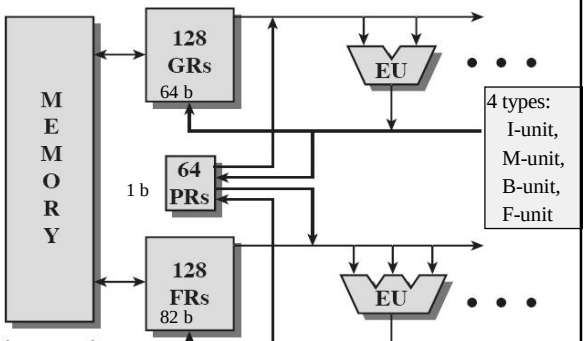
HOW to use?
Bigger cache ?
More processors?
More superscalar?

Billions?

Computer Organization II, Spring 2009, Tiina Niklander 20.4.2009 3

Itanium Organization

- Lot of registers => no renaming or dependency analysis
- Minimum 8 execution units
  - Function of the number of transistors available on chip
- GR-registers associated with NaT-bit (Not a Thing)
  - Used in speculations



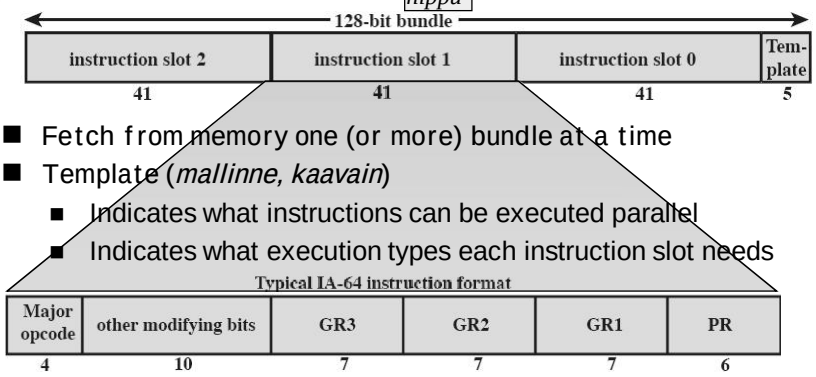
GR = General-purpose or integer register
FR = Floating-point or graphics register
PR = One-bit predicate register
EU = Execution unit

(Sta06 Fig 15.1)

Computer Organization II, Spring 2009, Tiina Niklander 20.4.2009 4

Instruction format

nippu



- Fetch from memory one (or more) bundle at a time
- Template (*mallinne, kaavain*)
  - Indicates what instructions can be executed parallel
  - Indicates what execution types each instruction slot needs
- PR: Instructions have predicate register (for speculative execution of instructions)
  - 1-bit, value checked at commit
- Instructions normally use 3 registers
- Load/Store -architecture

(Sta06 Fig 15.2)

Template and Execution units

(Sta06 Table 15.2, 15.3)

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-unit or M-unit
I	Non-ALU integer	I-unit
M	Memory	M-unit
F	Floating-point	F-unit
B	Branch	B-unit
X	Extended	I-unit/B-unit

- Template: info for parallel execution
  - Several bundles can be combined
    - More parallel execution
  - Black line = STOP (dependency)
  - No need for NOP instructions
- up to 6 instructions per clock cycle (source: Itanium data sheet)

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
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
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
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit

Assembly-language format (*symbolinen konekieli*)

`[qp] mnemonic[.comps] dests = srcs`

- qp qualifying predicate register
  - if predicate register value =1 (true), commit
- mnemonic name of the instruction operaation
- comps completers, separated by periods
  - Some instructions have extra parts to qualify it
- dests destination operands, separated by commas
- srcs source operands, separated by commas

Assembly-language format

- Instruction group boundaries (stops) are marked ;;
  - Instruction bundle template has the “black line”
  - Hint: the instructions of a group can be executed parallel
  - No data or output dependency within group
    - no read after write (RaW) or
    - no write after write (WaW)
  - What about antidependency (WaR)???

```
ld8 r1 = [r5]          // first group
sub r6 = r8, r9 ;;
add r3 = r1, r4        // second group
st8 [r6] = r12         // memory address in r6
```

Computer Organization II

Key mechanisms

- Predicated execution
- Control speculation (= speculative loading)
- Data speculation
- Software pipelining (*ohjelmoitu liukuhihna*)

Intel slides: http://www.cs.helsinki.fi/u/kerola/tikra/IA64-Architecture.pdf

Predicated execution (*Predikoitu suoritus*)

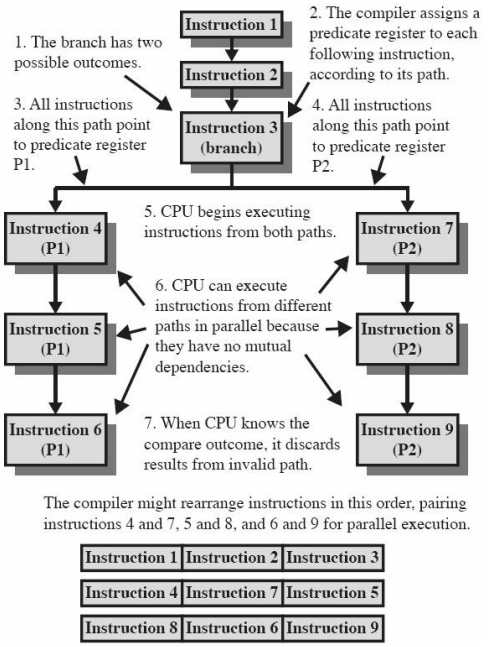
Compiler
- Create bundles, set template
  - Describe, which instructions can be executed parallel
  - Instruction execution order within one bundle undetermined
- Eliminate branches, e.g. if-then-else 'jumps'
  - Assign own predicate register to each branch
  - Both branches could be executed (in parallel?)

CPU

Intel slide 18
- Start executing both branches
  - Even before the condition result is known!
- Check predicates, when compare outcome known
  - Discard the results of the unselected branch
  - Commit the results ot the selected branch
  - Predicate always ready at instruction commit?

Predicated execution

- A graph about the execution
- Speculate:
  - Both branches executed until the right one can be selected
  - Selection known at the latest when the branch instruction (3) commits



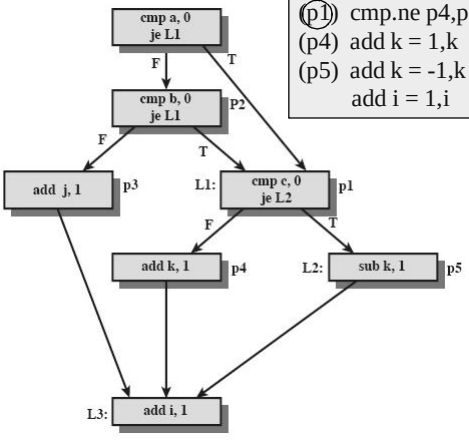
(Sta06 Fig 15.3a)

Predicated execution

Source:
```
if (a&&b)
  j=j+1
else
  if (c)
    k=k+1
  else
    k=k-1
i=i+1;
```

Pentium:
```
     cmp a,0
     je L1
     cmp b,0
     je L1
     add j,1
     jmp L3
L1:  cmp c,0
     je L2
     add k,1
     jmp L3
L2:  sub k,1
L3:  add i,1
```

IA-64:
```
     cmp.eq p1,p2 = 0,a ;;
(p2) cmp.eq p1,p3 = 0,b
(p3) add j = 1,j
(p1) cmp.ne p4,p5 = 0,c
(p4) add k = 1,k
(p5) add k = -1,k
     add i = 1,i
```



(Sta06 Fig 15.4)

Speculative loading

- Start data load in advance
  - = speculative load (even if unclear if the data is needed)
  - Ready at processor when needed, no latency
  - Straightforward unless branch or store between load and use
- Branching? – control speculation
  - Speculative loading could cause an exception (or page fault) that should not have happened at all
- Store? – data speculation
  - Speculative loading could be for the same memory location, the store is about to change

```
…
Comp   R1, =Limit
JLE    Done
Load   R5, Table(R1)
…
```

```
…
Store R1, (R3)
Load R5, (R4)
…
```


Intel slide 26

Control Speculation

- = Hoist (*nosta*) load instruction earlier in the code
  before the branch instruction
  - Mark it speculative (.s)
  - If speculative load cause exception, delay it (NaT bit)
    - There is a possibility that exception should not happen!
  - Add chk.s instruction to the original location. It checks for exceptions and starts recovery routine

```
je  L2
ld8  r1=[r5]
use r1
```

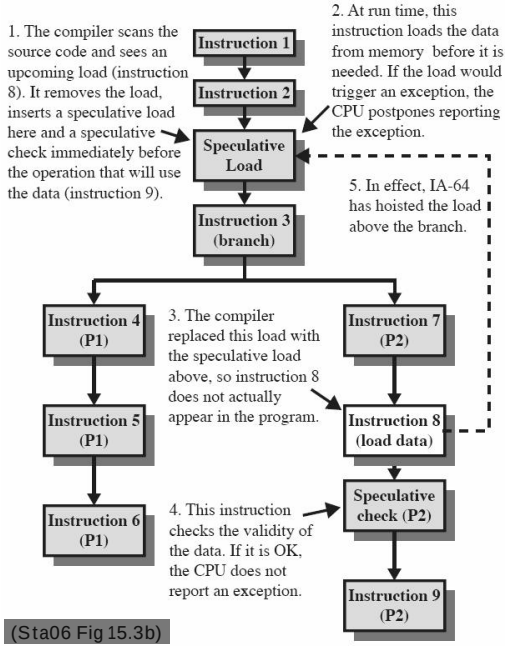
```
ld8.s  r1=[r5]
je  L2
chk.s  r1, recovery
use r1
```

← completer

Control speculation

- Compiler adds the speculative load earlier in the code and chk.s instruction in its original place
- CPU delays the (possible) exception from speculative load to the chk.s instruction

Intel slides 27-28



(Sta06 Fig 15.3b)

Intel slides 29-30

Data speculation

- = Hoist (*nosta*) load instruction earlier in the code
  before the store instruction
  - Mark is as advanced load (.a) (*ennakkolataaminen*)
  - Mark the original location with check (.c)
- Advanced Load Address Table (ALAT) hardware structure contains the memory addresses of current loads
  - Each load puts its memory location to ALAT
  - Each store removes its memory location from ALAT
  - Load check (.c): If target not in table, load again

```
je L1
st8 [r3] = r13
ld8  r1 = [r5]
```

⇒

```
ld8.a  r1 = [r5]
je L1
st8 [r3] = r13
ld8.c  r1 = [r5]
```

Aliasing issue:
r3 and r5 could point to same memory location

Intel slide 31

Software pipeline (*Ohjelmoitu liukuhihna*)

Why called software pipeline?

- Hardware support to allow parallel execution of loop instructions
- Parallel execution can be achieved by executing instructions of different iteration cycles
- Each iteration cycle uses different registers
  - Automatic register renaming
- Prolog (*alku*) and epilog (*loppu*) are special cases handled by rotating predicate register
- ”Loop jump” replaced by special loop termination instr. that controls the pipeline
  - Rotate registers, decrease loop count

Software pipeline

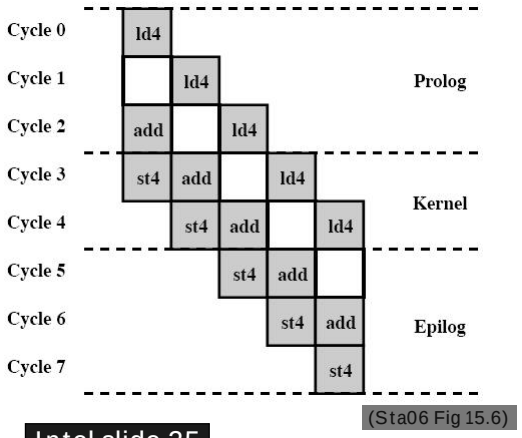
```
for i=5 to 1 do y[i] = x[i] + c
```

```
      mov lc = 5
L1: ld4   r4 = [r5],4 ;;
      add  r7 = r4,r9 ;;
      st4   [r6] = r7,4
      br.cloop L1 ;;
```

very little instruction-level parallelism, small code

A lot of instruction-level parallelism!
Operate different iteration regs at the same time

```
ld4  r32 = [r5], 4 ;;  // cycle 0
ld4  r33 = [r5], 4 ;;  // cycle 1
ld4  r34 = [r5], 4     // cycle 2
add r36 = r32, r9 ;; // cycle 2
ld4  r35 = [r5], 4     // cycle 3
add r37 = r33, r9     // cycle 3
st4  [r6] = r36, 4 ;;  // cycle 3
ld4  r36 = [r5], 4     // cycle 4
add r38 = r34, r9     // cycle 4
st4  [r6] = r37, 4 ;;  // cycle 4
add r39 = r35, r9     // cycle 5
st4  [r6] = r38, 4 ;;  // cycle 5
add r40 = r36, r9     // cycle 6
st4  [r6] = r39, 4 ;;  // cycle 6
st4  [r6] = r40, 4 ;;  // cycle 7
```



(Sta06 Fig 15.6)

Intel slide 25

```
        mov lc = 199                // set loop count register
        mov ec = 4                  // set epilog count register
        mov pr.rot = 1<<16;;        // pr16 = 1, rest = 0
L1:  (p16) ld5 r32 = [r5], 4        // cycle 0
     (p17) ---                      // empty stage
     (p18) add r35 = r34, r9        // cycle 0
     (p19) st4 [r6] = r36, 4        // cycle 0
        br.ctop L1 ;;               // cycle 0
```

Koodi p.555

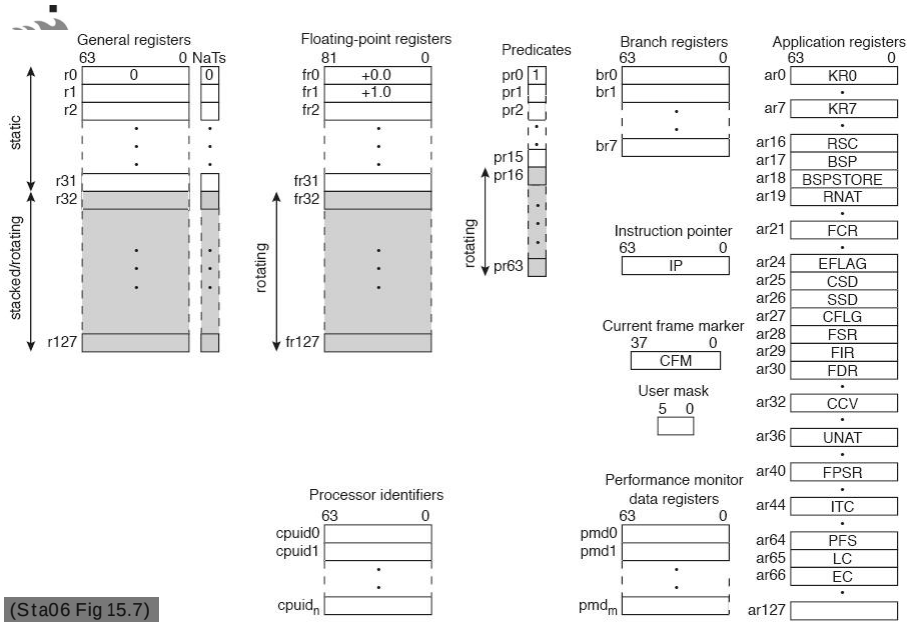
Code

(Sta06 Table 15.4)

Cycle	Execution Unit/Instruction				State before br.ctop					
	M	I	M	B	p16	p17	p18	p19	LC	EC
0 prolog	ld4			br.ctop	1	0	0	0	199	4
1	ld4			br.ctop	1	1	0	0	198	4
2	ld4	add		br.ctop	1	1	1	0	197	4
3	ld4	add	st4	br.ctop	1	1	1	1	196	4
•••	•••	•••	•••	•••	•••	•••	•••	•••	•••	•••
100	ld4	add	st4	br.ctop	1	1	1	1	99	4
•••	•••	•••	•••	•••	•••	•••	•••	•••	•••	•••
199	ld4	add	st4	br.ctop	1	1	1	1	0	4
200		add	st4	br.ctop	0	1	1	1	0	3
201 epilog		add	st4	br.ctop	0	0	1	1	0	2
202			st4	br.ctop	0	0	0	1	0	1
					0	0	0	0	0	0

Computer Organization II

Itanium Registers



(Sta06 Fig 15.7)

Application register set (*Sovelluksen rekisterit*)

Sta06 Fig 15.7

- General registers(128), FP-registers (128), Predicates (64)
  - Some static, some rotational (automatic renaming by hw)
  - Some general registers used a stack (*pino*)
- Branch registers (8)
  - Target address can be in a register (indirect jump!)
  - Subroutine return address normally stored in register br0
  - If ner call before return, br0 stored in register stack
- Instruction pointer
  - Bundle address of current instruction
  - not address of single instruction
- User mask
  - Flags (single-bit values) for traps and performance monitoring
- Performance monitor data registers
  - Supports monitoring hardware
  - Information about hardware, e.g. branch predictions, usage of register stack, memory access delays, …

Intel slides 15-17

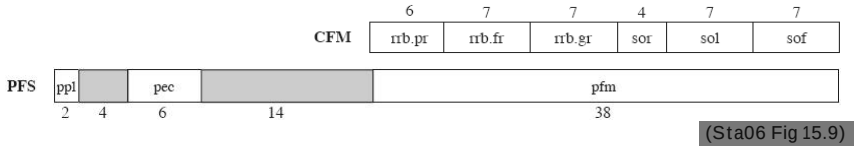
Rekisteripino, Register Stack Engine

- r0..r31 for global variables
- r32..r127 (total 96) for subroutine calls
- Call reserves a frame (set of regs in a register window)
  - parameters (inputs/outputs) + local variables
  - Size set dynamically (alloc instruction)
- Registers automatically renamed after the call
  - Subroutine parameters always start from register r32
- Allocated in a circular-buffer fashion (*renkaana*)
  - If area full, hardware moves register contents of oldest frame to memory (= backing store). Restored when subroutine returns
    - Memory address in register BSP, BSPSTORE (backing store pointer)

Intel slide 17

Register stack

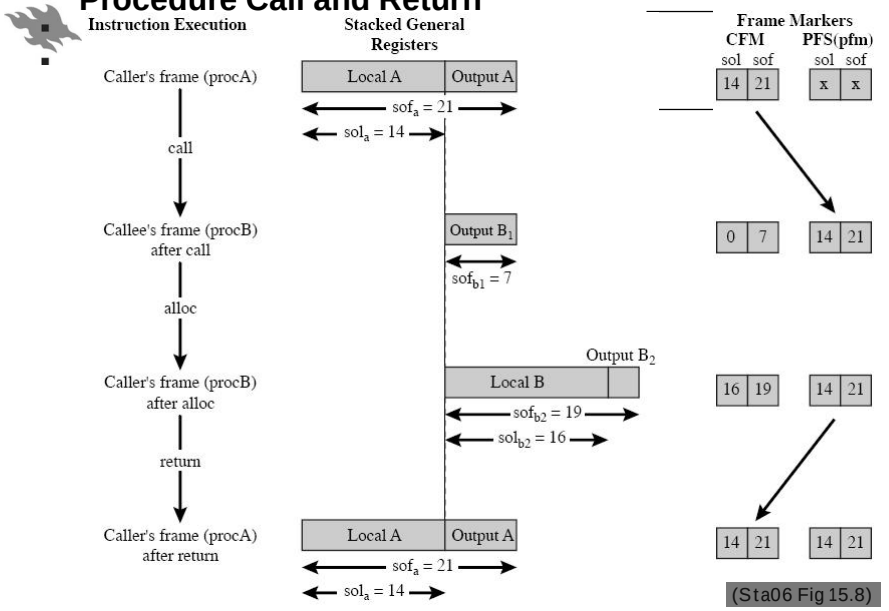
- Allocation and restoring using two dedicated registers
- CFM, Current Frame Marker
  - Size of the most-recently allocated area
    - sof=size of frame, sol=size of locals,
    - sor=size of rotation portion (SW pipeline)
  - GR/FP/PR register rotation information
    - rrb=register rename base
- PFS, Previous Function State
  - Previous value of CFM stored here. Older content of PFS stored somewhere else (another register?) (alloc determines the destination)



(Sta06 Fig 15.9)

Intel slide 17

Procedure Call and Return



(Sta06 Fig 15.8)

(Sta06 Table 15.5)

Application registers

Kernel registers (KR0-7)	Convey information from the operating system to the application.
Register stack configuration (RSC)	Controls the operation of the register stack engine(RSE)
RSE Backing store pointer (BSP)	Holds the address in memory that is the save location for r32 in the current stack frame.
RSE Backing store pointer to memory stores (BSPSTORE)	Holds the address in memory to which the RSE will spill the next value.
RSE NaT collection register (RNAT)	Used by the RSE to temporarily hold NaT bits when it is spilling general registers.
Compare and exchange value (CCV)	Contains the compare value used as the third source operand in the cmpxchg instruction.
User NaT collection register (UNAT)	Used to temporarily hold NaT bits when saving and restoring general registers with the ld8.fill and st8.spill instructions.
Floating-point status register (FPSR)	Controls traps, rounding mode, precision control, flags, and other control bits for floating-point instructions.
Interval time counter (ITC)	Counts up at a fixed relationship to the processor clock frequency.
Previous function state (PFS)	Saves value in CFM register and related information.
Loop count (LC)	Used in counted loops and is decremented by counted-loop-type branches.
Epilog count (EC)	Used for counting the final (epilog) state in modulo-scheduled loops.

Computer Organization II

Itanium 2

(again just called Itanium!)

Itanium

- First implementation released in 2001
- Second, **at that time called Itanium 2**, released in 2002
- Simpler than conventional superscalar CPU
  - No resource reservation stations
  - No reorganization buffers (ROB)
  - Simpler register remapping hardware (versus register aliasing)
  - No dependency-detection logic
  - Compiler solved dependences and created /computed explicit parallelism directives
- Large address space (*suuri osoiteavaruus*)
  - Smallest addressable unit: 1, 2, 4, 8, 10, 16 bytes
  - recommendation: use natural boundaries
- Support both Big-endian and Little-endian

Itanium

- Wide and fast bus: 128b, 6.4 Gbps
- Improved cache hierarchy
  - L1: split instr, data 16KB + 16KB, set-ass. (4-way), 64B line
  - L2: shared 256KB, set-ass. (8-way), 128B line
  - L3: shared, 3MB, set-ass. (12-way), 64B line
  - All on-chip, smaller latencies
- TLB hierarchy
  - I-TLB L1: 32 items, associative
  - L2: 128 items, associative
  - D-TLB L1: 32 items, associative
  - L2: 128 items, associative

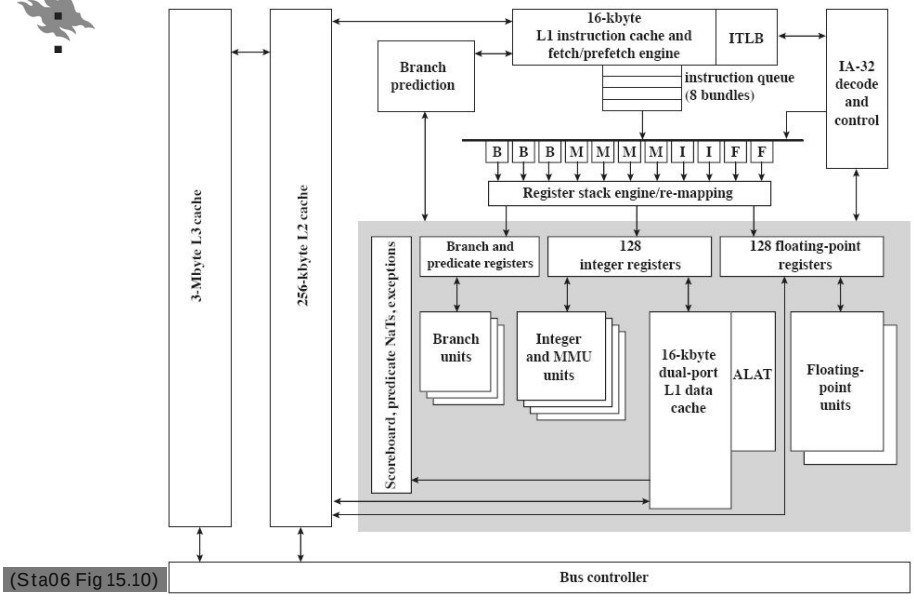
Memory management

- Memory hierarchy visible to applications also
- = possibility to give hints
  - Fetch order: make sure, that earlier ops have committed
  - Locality: fetch a lot / a little lines to cache
  - Prefetch: when moved closer to CPU
  - Clearing: line invalidation, write policy
- Implicit control (exclusive access)
  - Switching memory and register content
  - Increasing memory content by a constant value
- Possibility to collect performance data
  - To improve hints…

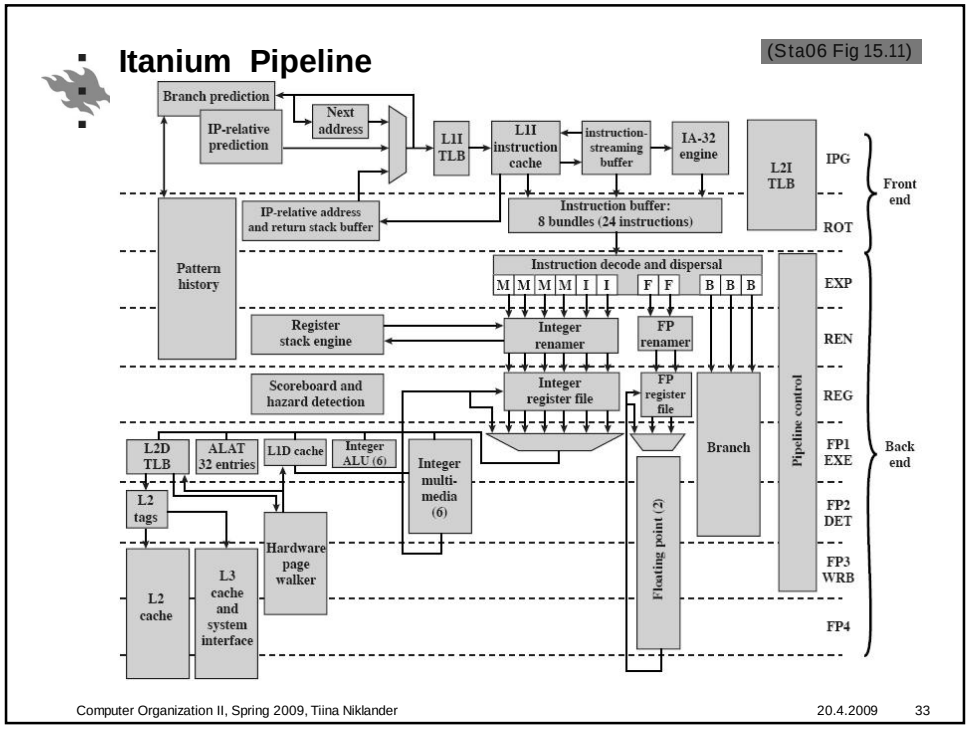
Itanium

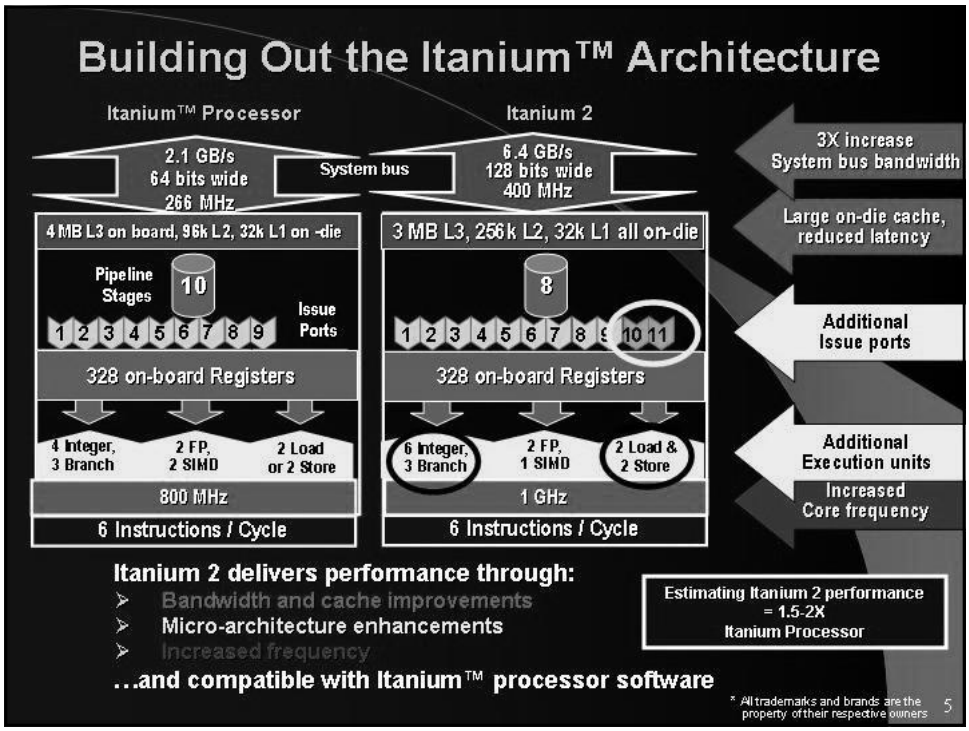
- 11 instruction issue port (like selection window)
- Max 6 instructions to execution in each cycle
  - in-order issue, out-of-order completion
- 8-stage pipeline
- More execution units (22)
  - 6 general purpose ALU's (1 cycle)
  - 6 multimedia units (2 cycles)
  - 3 FPU's (4 cycles)
  - 3 branch units
  - 4 data cache memory ports (L1: 1/2 cycle load)
- Improved branch prediction
  - Application is allowed to give hints
  - Used to recude cache miss

Itanium Processor



(Sta06 Fig 15.10)


Itanium Pipeline
(Sta06 Fig 15.11)
Branch prediction
Next address
IP-relative prediction
L1I TLB
L1I instruction cache
instruction-streaming buffer
IA-32 engine
L2I TLB
IPG
Front end
IP-relative address and return stack buffer
Instruction buffer: 8 bundles (24 instructions)
ROT
Pattern history
Instruction decode and dispersal
M M M M I I F F B B B
EXP
Register stack engine
Integer renamer
FP renamer
REN
Scoreboard and hazard detection
Integer register file
FP register file
REG
Branch
Pipeline control
L2D TLB
ALAT 32 entries
L1D cache
Integer ALU (6)
Integer multi-media (6)
FP1 EXE
Back end
L2 tags
FP2 DET
Hardware page walker
Floating point (2)
L2 cache
L3 cache and system interface
FP3 WRB
FP4
Computer Organization II, Spring 2009, Tiina Niklander
20.4.2009
33


Building Out the Itanium™ Architecture
Itanium™ Processor
Itanium 2
2.1 GB/s
64 bits wide
266 MHz
System bus
6.4 GB/s
128 bits wide
400 MHz
3X increase
System bus bandwidth
4 MB L3 on board, 96k L2, 32k L1 on -die
3 MB L3, 256k L2, 32k L1 all on-die
Large on-die cache, reduced latency
Pipeline Stages
10
8
Issue Ports
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10 11
Additional Issue ports
328 on-board Registers
328 on-board Registers
4 Integer, 3 Branch
2 FP, 2 SIMD
2 Load or 2 Store
6 Integer, 3 Branch
2 FP, 1 SIMD
2 Load & 2 Store
Additional Execution units
800 MHz
1 GHz
Increased Core frequency
6 Instructions / Cycle
6 Instructions / Cycle
Itanium 2 delivers performance through:
Bandwidth and cache improvements
Micro-architecture enhancements
Increased frequency
Estimating Itanium 2 performance = 1.5-2X Itanium Processor
…and compatible with Itanium™ processor software
* All trademarks and brands are the property of their respective owners
5

Computer Organization II

Current State (2006-08)

Intel hyper-thread and multi-core

STI multi-core

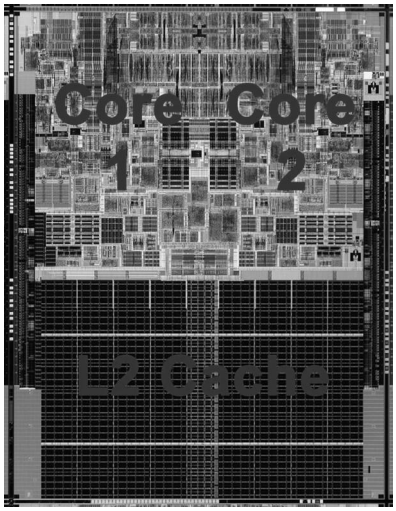
Intel Pentium 4 HT (IA-32)

- HT – Hyper-threading
- 2 logical processors in one physical prosessor
- OS sees it as symmetric 2-processor system
- Use wait cycles to run the other thread
  - memory accesses (cache miss)
  - dependencies, branch miss-predictions
- Utilize usually idle int-unit, when float unit in use
- 3.06 GHz + 24%(?)
  - GHz numbers alone are not so important
- 20 stage pipeline
- Dual-core hyper-thread processor
  - Dual-core Itanium-2 with Hyper-threading

http://www.intel.com/multi-core/index.htm

Intel Multi-Core Core-Architecture

- 2 or more (> 100?) complete cores in one chip
  - Hyper-threading still in use
  - Simpler structure, less power
  - Private L1 cache
  - Private or shared L2 cache?
- Intel Core 2 Duo E6700
  - 128-bit data path
  - Private 32 KB L1 data cache
  - Private 32 KB L1 instr. Cache (for micro-ops)
  - Shared/private 4 MB L2 data cache



http://www.hardwaresecrets.com/article/366

Click 1 or 2 for *Torres* articles

Click for *Pawlowski* article

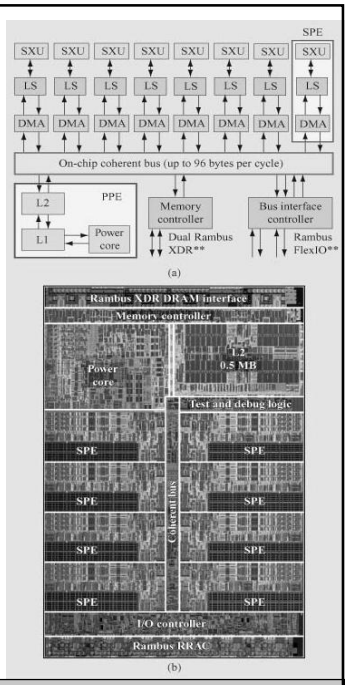
Computer Organization II

STI Cell Broadband Engine

(Sony-Toshiba-IBM)

STI Cell Broadband Engine

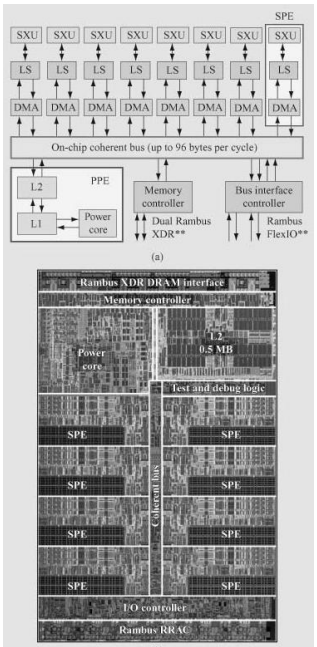
- Several dedicated vector-processing engines (SPE), controlled by one main, general purpose processor (PPE)
- 1 PowerPC PPE
  - Power Processing Element
  - RISC, 2 hyper-threads, in-order, simple prediction logic (needs compiler support)
  - 32 KB L1 data and instr. caches
  - 256KB L2 cache
  - "normal programs"
- 8 SPE's
  - Synergistic Processor Elements
  - 256KB local data/instr memory, no cache
  - Receive code/data packets from off-chip main memory (as DMA transfer)
  - 128 registers a' 128 bit, 64 GB/s
  - 2 pipelines: even, odd
  - No branch prediction, "branch hint"-instr.



http://researchweb.watson.ibm.com/journal/rd/494/kahle.html

STI Cell Broadband Engine

- Programming Models for SPE use
  - Function offload Model
    - Run some functions at SPE's
  - Device Extension Model
    - SPE as front-end for some device
  - Computational Acceleration Model
    - SPE's do most of computation
  - Streaming Models
    - Data flow from SPE to SPE
  - Shared-mem multiprocessor Model
    - Local store as cache
    - Cache coherent shared memory
  - Asymmetric Thread Runtime Model

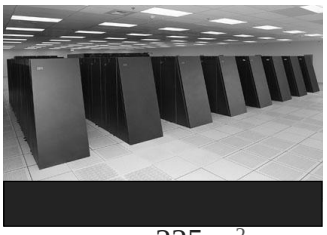


Click for *Kahle et al* article

http://researchweb.watson.ibm.com/journal/rd/494/kahle.html

STI Cell (Cell B.E.)

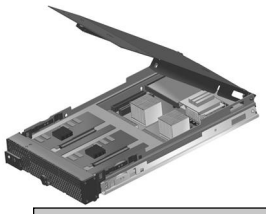
- Sony
  - Playstation 3 (4 cells)
- IBM
  - Roadrunner supercomputer (installed 2008)
    - $110M, 1100 m², Linux
    - Peak 1.6 petaflops (1.6 * 10^{15} flops)
      - Sustained 1 petaflops
    - Over 16000 AMD Opterons for file ops and communication (e.g.)
      - Normal servers
    - Over 16000 Cells for number crunching
      - Blade centers



processors, 225 m^2

STI Cell (Cell B.E.)

- Toshiba
  - Quad Core HD processor (or SpursEngine)
    - In multimedia laptops for HD DVD's
- Mercury Computer Systems (year 2006)
  - Cell accelerator board (CAB) for PC's
  - 180 GFlops boost, Linux
- Blade servers
  - Mercury 42U Dual Cell Based Blade 2 Systems
    - 42 Dual Cell BE Processors
  - IBM BladeCenter
    - 2 IBM PowerXCell 8i processor



Mercury Dual-Cell Blade

IBM Blade Server prototype w/ 2 cells (2005)

Computer Organization II

ARM -

ARM architecture family

- 32-bit embedded RISC microprocessor
  - ARM family accounts for approximately 90% of these.
  - The most widely used 32-bit CPU architecture in the world.
  - The ARM architecture is used in about 3/4 of all 32 bit processors sold. (source Intellitech)
  - Exists in 95% of all cell phones (source: Intellitech)
- Architecture
  - Extremely simple (ARM6 only 35,000 transistors)
    - 32-bit data bus, a 26-bit (64 Mbyte) address space and sixteen 32-bit registers.
    - **low power usage**, hardwired-control
  - ARM: no cache, ARM4: cache
  - ARM8: 5-stage pipeline, static branch prediction, double-bandwidth memory

ARM architecture

- Conditional execution of most instructions
  - 4-bit condition code in front of every instruction
- Arithmetic instructions alter condition codes only when desired
- Indexed addressing modes
- 2-priority-level interrupt subsystem

```
while (i != j)
{ if (i > j)
  i -= j;
else j -= i; }
```

```
loop CMP Ri, Rj  ; set condition "NE" if (i != j)
                 ;                "GT" if (i > j),
                 ;             or "LT" if (i < j)
SUBGT Ri, Ri, Rj ; if "GT", i = i-j;
SUBLT Rj, Rj, Ri  ; if "LT", j = j-i;
BNE loop         ; if "NE", then loop
```


Things progress…

- X86 => Pentium => Core => Nehalem (Core i7) => Westmere
  - Superscalar
    - More efficient use of pipelining
      - Parallel pipelines
      - Branch prediction
      - Out-of order -execution
      - CISC => RICS translations
      - Hyperthreading
  - Chip –level multiprocessing -> multi-core
  - Vector instruction codes (in vector processors)
    - Parallel data processing
  - Cache: more levels, larger cache
    - OX9650: 12 MB L2

To different directions ...

- Power consumption (*Virrankulutus*)
  - Mobile and portable devices
  - density => heating up
- Superscalar improvements used?
  - Improments prediction logic gives less and less benefit => simpler CPU
    - => software based (Transmetan Crusoe tried this!)
    - => compiler does and gives better ordered instructions (IA-64, Itanium2, CELL, ..)
- More cores on one chip
  - Different tasks (like Westmere integrated GPU)
  - Coordination of the cores (processors) becomes an issue

Review Questions / Kertauskysymyksiä

- EPIC?
- Why does the instruction bundle have a template?
- What is predicated execution? How does it work?
- What means control speculation? Data speculation?
- How registers are used in subroutine calls?
- Difference of hyper-threading and multi-core?

- EPIC?
- Miksi käskynipun yhteydessä on template?
- Mitä tarkoitetaan predikoinnilla? Kuinka se toimii?
- Mitä tarkoittaa kontrollispekulointi? Entä dataspekulointi?
- Miten rekistereitä käytetään aliohjelmakutsuissa?
- Mikä ero hyper-threadeillä ja multi-corella?