ITANIUM
An EPIC Architecture

Marco Barcella
Karthik Sankaranarayanan
Ganesh Pai

Introduction

- EPIC: Explicitly Parallel Instruction Computing
- Combination of features of RISC and VLIW
- VLIW features and flaws
  - Groups of independent instructions
  - Simple hardware
  - Exploit ILP with compiler
  - Large increase in code size
  - Blocking caches
Introduction

- 733 - 800 MHz clock
  0.18-micron CMOS process technology

- 2 extended, 2 single precision FMACs
  Execution up to 8 SP flops/cycle - 6 GFLOP

- >20x Pentium Pro

- 3-level cache hierarchy
  - Split L1 and Unified L2 on die
  - Unified L3 on separate die but same container

Introduction

- 64-byte line size
- Page Sizes up to 256MB
- Full 64-Bit computing
- Full IA-32 binary compatibility in hardware
  - Shared Resources: ALU, registers, Data Cache
  - IA-32 Engine: Dynamic execution

- Instruction set architecture (Marco)
  Instruction stream (Ganesh)
  Data stream and IA-32 Compatibility (Karthik)
Die Plot

Instruction Set Architecture
The Software Interface

Marco Barcella
Outline

- Introduction to the ISA
- Expressing parallelism
- Creating parallelism
- Techniques and instructions
- Compatibility
- Observations

Why & How

- Goal
  - Bring ILP features to a general purpose microprocessor, flexibility

- Techniques
  - Predication
  - Speculation
  - Large register files
  - Register rotation
  - HW exception deferral
  - Software pipelining

- RISC/CISC basic architecture of HP’s PA-RISC, but ...
## Register Resources

- NaT
- Predicate Registers
- Branch Registers

- AR: PFS(PFM, PEC)
  - UM
  - 8 Kernel Registers
  - LC, EC, CCV
  - AR 16-19
  - Future definition

## Encoding

- Bundles: More than one per cycle
- Template: MII, MIB – other combinations
- Compiler based reordering
- No Register analysis
- Instruction compared to 32-bit
Instructions

- 6 types, 4 units
- L+ X : Long branches, long immediate integer

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Description</th>
<th>Execution Unit Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Integer ALU</td>
<td>I-unit or M-unit</td>
</tr>
<tr>
<td>I</td>
<td>Non-ALU integer</td>
<td>I-unit</td>
</tr>
<tr>
<td>M</td>
<td>Memory</td>
<td>M-unit</td>
</tr>
<tr>
<td>F</td>
<td>Floating-point</td>
<td>F-unit</td>
</tr>
<tr>
<td>B</td>
<td>Branch</td>
<td>B-unit</td>
</tr>
<tr>
<td>L+X</td>
<td>Extended</td>
<td>I-unit/B-unit</td>
</tr>
</tbody>
</table>

Expressing Parallelism

- Not only bundles, but also
- Compound Conditionals

```c
If ((a==0) || (b<=5) || (c!=d) || (f & 0x2)) { r3 = 8; };
```

```c
cmp.ne pl = r0, r0;
add t = -5, b;;
cmp.eq.or pl = 0, a
cmp.ge.or pl = 0, t
cmp.ne.or pl = 0, d
tbit.or pl = 1, f, 1;;
(p1) mov r3 = 8
```

- Multi-way branches

```c
{ .mi
  cmp.ne p1,p2 = r1,r2;
  cmp.ne p3,p4 = 4, r5;
  cmp.lt p5, p6 = r8,r9;
}
{ .bbb
  (p1) br.cond label1
  (p3) br.cond label2
  (p5) br.call b4 = label3
}
// Fall through code here
```
Creating Parallelism

- **Predication**
  - Uses CMP instructions and predicate registers
  - Converts control dependencies to data dependencies
  - Motivation
    
    ```
    if (r1==r2)
      r9 = r10 - r11;
    else
      r5 = r6 + r7;
    ```

    ```
    cmp.eq p1,p2 = r1, r2;
    (p1) sub r9 = r10, r11
    (p2) add r5 = r6, r7
    ```

- **Speculation + Predication**
  - Basic blocks in a single group
  - Barriers between basic blocks
  - Compiler

Control Speculation

- Importance of loads
- ld.s and chk.s and handling exceptions
- Propagation of token and fix-up
Data Speculation

- Ambiguous dependencies, ld.a

- How it works
  - ALAT, two tags

- Two recoveries
  - ld.c, ldf.c, ldfp.c
  - chk.a (chk.s)

Procedure Calls

- Criticism: Large registers
- GR: 32 static + 96 stack
- Frames(SPARC), local, output
- br.call, br.l.call & then br.ret
  - CFM in PFM (PFS), RRB, alloc (sof, sol)
**Procedure Calls**

- RSE speculatively fills and spills in the background
- Result: Vs. PA-RISC 30%, 5% (Database)

**Context Switch Instructions**

- Specific control on stack and backing store
  - Flushrs to spill previous stack frames
  - Cover to create a new frame above
  - Ladrs to fill from backing store
Branch Instruction

- Three categories
  - IP-relative (21 bit); Long (60 bit); Indirect (in BRs)

Table 4-23. State Relating to Branching

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRs</td>
<td>Branch registers</td>
</tr>
<tr>
<td>PRs</td>
<td>Predicate registers</td>
</tr>
<tr>
<td>CPM</td>
<td>Current Frame Marker</td>
</tr>
<tr>
<td>PFS</td>
<td>Previous Function State application register</td>
</tr>
<tr>
<td>LC</td>
<td>Loop Count application register</td>
</tr>
<tr>
<td>EC</td>
<td>Epilog Count application register</td>
</tr>
</tbody>
</table>

Figure 3-2. Frame Marker Format

Table 4-22. Branch Types

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Branch Condition</th>
<th>Target address</th>
</tr>
</thead>
<tbody>
<tr>
<td>be.cond or be</td>
<td>Conditional branch</td>
<td>Qualifying predicate</td>
<td>IP-rel or Indirect</td>
</tr>
<tr>
<td>be.call</td>
<td>Conditional procedure call</td>
<td>Qualifying predicate</td>
<td>IP-rel or Indirect</td>
</tr>
<tr>
<td>be.mak</td>
<td>Conditional procedure return</td>
<td>Qualifying predicate</td>
<td>Indirect</td>
</tr>
<tr>
<td>be.ie</td>
<td>Invoke the IA-32 instruction set</td>
<td>Unconditional</td>
<td>Indirect</td>
</tr>
<tr>
<td>be.cmp</td>
<td>Counted loop branch</td>
<td>Loop count.</td>
<td>IP-rel</td>
</tr>
<tr>
<td>be.cmp, be.cestz</td>
<td>Module-scheduled counted loop</td>
<td>Loop count and Eiplog count</td>
<td>IP-rel</td>
</tr>
<tr>
<td>be.cep, be.cestz</td>
<td>Module-scheduled while loop</td>
<td>Qualifying predicate and Eiplog count</td>
<td>IP-rel</td>
</tr>
<tr>
<td>be.cond on be</td>
<td>Long conditional branch</td>
<td>Qualifying predicate</td>
<td>IP-rel</td>
</tr>
<tr>
<td>be.call</td>
<td>Long conditional procedure call</td>
<td>Qualifying predicate</td>
<td>IP-rel</td>
</tr>
</tbody>
</table>
Software Pipelining

- Motivation
- Vs HW
- Parallelism
- 3 phases
- Rotating FR, PR
- LC, EC

Software Pipelining

- 2 categories
  - Counted,
  - While (top, exit)

- Counted
  - Ends with EC=1 and LC=0, no qualifying predicate

- While
  - No LC, ends when QP=0 and EC=1
Branch Prediction Hints

- Hints, Branch Predict Instructions (brp)
- Hints:
  - strategy

<table>
<thead>
<tr>
<th>Completor</th>
<th>Strategy</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>hint</td>
<td>Static not-Taken</td>
<td>Ignore the branch, do not allocate prediction resources for this branch.</td>
</tr>
<tr>
<td>hint</td>
<td>Static Taken</td>
<td>Always predict taken, do not allocate prediction resources for this branch.</td>
</tr>
<tr>
<td>dyn</td>
<td>Dynamic Not-Taken</td>
<td>Use dynamic prediction hardware. If no dynamic history information exists for this branch, predict not-taken.</td>
</tr>
<tr>
<td>dyn</td>
<td>Dynamic Taken</td>
<td>Use dynamic prediction hardware. If no dynamic history information exists for this branch, predict taken.</td>
</tr>
</tbody>
</table>

Branch Prediction Hints

- Prefetch

<table>
<thead>
<tr>
<th>Completor</th>
<th>Sequential Prefetch Hint</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>few</td>
<td>Prefetch few lines</td>
<td>When prefetching code at the branch target, stop prefetching after a few (implementation-dependent number) lines.</td>
</tr>
<tr>
<td>many</td>
<td>Prefetch many lines</td>
<td>When prefetching code at the branch target, prefetch many lines (also an implementation-dependent number).</td>
</tr>
</tbody>
</table>

- Deallocate

<table>
<thead>
<tr>
<th>Completor</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>Don’t deallocate</td>
</tr>
<tr>
<td>ok</td>
<td>Deallocate branch information</td>
</tr>
</tbody>
</table>
Branch Prediction Hints

- Branch prediction instructions
  - LOCATION
  - TARGET
  - IMPORTANCE
  - STRATEGY

Memory Instructions

- Simple (GR or FR, memory access order)
- Variants for speculative, spilling
- Semaphore instructions
Memory Instructions

Table 4-12. Memory Access Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Floating-point</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>li</td>
<td>load</td>
<td>Load</td>
</tr>
<tr>
<td>lw, lwi</td>
<td>load</td>
<td>Speculative load</td>
</tr>
<tr>
<td>1d</td>
<td>1dp</td>
<td>Advanced load</td>
</tr>
<tr>
<td>1d.d</td>
<td>1dp.d</td>
<td>Speculative advanced load</td>
</tr>
<tr>
<td>1d.c.m, 1d.c.c</td>
<td>1dp.c.m, 1dp.c.c</td>
<td>Check load</td>
</tr>
<tr>
<td>1d.c.c,m</td>
<td></td>
<td>Ordered check load</td>
</tr>
<tr>
<td>1d.mq</td>
<td></td>
<td>Ordered load</td>
</tr>
<tr>
<td>1d.mh/m</td>
<td></td>
<td>memory load</td>
</tr>
<tr>
<td>1d régl</td>
<td>registro</td>
<td>Register fill</td>
</tr>
<tr>
<td>1w</td>
<td>store</td>
<td>Store</td>
</tr>
<tr>
<td>1w, rel</td>
<td></td>
<td>Order store</td>
</tr>
<tr>
<td>1w, spill</td>
<td></td>
<td>Register spill</td>
</tr>
<tr>
<td>mem.dex</td>
<td></td>
<td>Compare and exchange</td>
</tr>
<tr>
<td>xchg</td>
<td></td>
<td>Exchange memory and GPR</td>
</tr>
<tr>
<td>fetchd</td>
<td></td>
<td>Fetch and add</td>
</tr>
</tbody>
</table>

Integer and Shifting

- Add, add...1, addp (32bit)

- Shift Left Mask Merge: dep, dep.z
  - Position and field by immediate
  - Simple shl (amount)
**Compare Instructions**

- Two predicate registers
- Deferred token (tnat)
- 5 types
  - Normal,
  - Unconditional
  - 3 "parallel" compares

**Floating Point Architecture**

- FSR: precision modes, 4 status fields
- All with FMAC= A*B+C: simple, divide
- XMA
- 82 bits: 2+ 32(if single), 64(double), 80(double extended)
- Two singles in one register
Compatibility

- X86: direct execution
- BR.1A, JMPE, overhead of register set saving
- SSE included (128), "new media"
- MMX parallel arithmetic: 128 not 8
- HP dynamic translator
- CMP4

Code Density

- Causes
  - Avg. 43 bit (32 of RISC)
  - Added (alloc, chk)
  - Fix-up
- Biggest impact
  - Decreasing hit rate on caches
Observations

- Synergetic
  - ld.sa, data dependences in software pipelining

- Compiler
  - Template
  - Grouping
  - Explicit prefetching
  - ld.a

- X86 common SW base (aggressive)

- 20/30% improvement over RISC is claimed

Instruction Stream

The Processor Front-end

Ganesh Pai
Instruction Stream

- Overview of EPIC hardware

- I-Stream
  - Pipeline
  - I-Cache
  - Prefetch & Fetch
  - Branch prediction
  - Issue (Instruction dispersal & delivery)

Overview of EPIC Hardware

<table>
<thead>
<tr>
<th>Compiler-programmed features:</th>
<th>Explicit parallelism; instruction templates</th>
<th>Register stack, rotation</th>
<th>Prediction</th>
<th>Data and control speculation</th>
<th>Memory hints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch hints</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hardware features:</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction cache, branch predictors</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register handling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>128 GR, 128 FR, register remap, stack engine</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Control</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel resources</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 integer, 4 MMX units</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data and control speculation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory subsystem</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Three levels of cache (L1, L2, L3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Speculation deferral management
**10 Stage In-order Core Pipeline**

**Front End**
- Pre-fetch/Fetch of up to 6 instructions/cycle
- Hierarchy of branch predictors
- Decoupling buffer

**Execution**
- 4 single cycle ALUs, 2 ld/st
- Advanced load control
- Predicate delivery & branch
- Nat/Exception/Retirement

**Pipeline Features**

- 6-wide EPIC hardware under precise compiler control
- 10-stage in-order pipeline
- Dynamic support for run-time optimization
  - Ensure high throughput
- Register scoreboard to enforce dependencies
I – Cache; I – TLB

- 16 Kb
- 4-way set associative
- Fully pipelined
- 64-entry I-TLB
- Single cycle
- Fully associative
- On-chip page walker

- I-Cache filters prefetch requests
- Both enhanced with an additional port
  - To check for a miss

Fetch & Prefetch

- Speculative fetching

- Both hardware and software prefetching

- Software initiated instruction prefetch
  - Triggered by BPR hints
  - Fetch from L2 into instruction-streaming buffer (ISB)
  - Eight 32-byte entries in the ISB
  - Short 64-byte bursts / long sequential stream

- Eliminate I-fetch bubbles
Fetch & Prefetch

- Decoupling buffer
  - 8 bundles deep
  - Hides stalls, cache misses, branch mispredictions

Branch Prediction

- First emphasis on compiler
  - Reducing branches by predication

- Branch Prediction for remaining cases
  - Assisted by branch hint directives i.e
  - branch target addresses
  - Static hints on branch direction
  - Indications for use of dynamic predictor

- Hierarchy of branch predictors
Branch Prediction

- Branch hints + Predictor Hierarchy
  - Four progressive Resteers
  - Improved branch prediction

Branch Prediction

- Resteer1: Single Cycle Predictor
  - 4 TARs programmed by compiler with “important” hints
  - TAR is a 4 deep FIFO
  - On a “hit” branch is predicted taken

- Resteer2: Adaptive multi-way return predictor
  - 2 level prediction scheme (Yeh and Patt)
  - 512 (128 x 4) entry branch prediction table (BPT)
  - 2 bit saturating up-down counter to predict direction
  - Enhanced by 64-entry multi-way BPT
  - 64-entry branch target address cache (BTAC)
  - 8-entry return stack buffer (RSB)
**Branch Prediction**

- Resteer3 & 4
  - Two branch address calculators (BAC1 and BAC2)
  - Correction to earlier predictions (if any)
  - A special “perfect-exit-loop-predictor”

- In case of misses in earlier structures
  - Use of a static prediction information from bundles

**Instruction Dispersal**
### Instruction Dispersal

- Stop bits eliminate dependency checking
- Templates simplify routing
- Map instructions to first available of 9 issue ports
  - Keep issuing until stop bit
  - Resource over-subscription or asymmetry
- Re-map virtual register to physical register
- Instruction granular

### Instruction Delivery

- Register Stacking
- Achieved transparently to the compiler
- Register re-mapping via parallel adders
Data Stream

The Execution Core

Karthik Sankaranarayanan

Recap - Execution Units

- 17 units + ALAT
  - 4 ALU
  - 4 MMX
  - 2 + 2 FMAC
  - 2 Load/ Store
  - 3 branch

- Issue Ports
  - 2 I
  - 2 M
  - 2 F
  - 3 B
Register Files

- Integer
  - 128 64-bit
  - 8 read ports (2 x 2 I units, 2 x 2 M units)
  - 6 write ports (1 x 2 I units, 2 x 2 Loads - A.I)

- Floating Point
  - 128 82-bit (double extended)
  - 8 read ports (2 x 2 F units, 2 x 2 M units)
  - 4 write ports (2 x 2 F units, 2 x 2 M units)

- Predicate
  - 64 1-bit, "broadside" R/W
  - 15 read ports (2 x 6 - M, F, I units & 3B units)
  - 11 write ports
    - (2 x 2 M units, 2 x 2 I units, 2 x 1 F unit, 1 x 1 Reg. Rot.)

Recap - 10 Stage Pipeline

Front End
- Pre-fetch/Fetch of up to 6 instructions/cycle
- Hierarchy of branch predictors
- Decoupling buffer

Execution
- 4 single cycle ALUs, 2 ld/str
- Advanced load control
- Predicate delivery & branch
- Nat/Exception/Retirement

Instruction Delivery
- Dispersal of up to 6 instructions on 9 ports
- Reg. remapping
- Reg. stack engine

Operand Delivery
- Reg read + Bypasses
- Register scoreboard
- Predicated dependencies
Operand Delivery - WLD/REG Stages

- **Register Read**
  - WLD (Word Line Decode) - begin access
  - REG - Read Registers
  - WLD - frequency increase?

- **Register Scoreboard**
  - Hazard detection
  - Stall only dependent instructions
  - Include predicates

\[
\begin{align*}
\text{cmp.eq } r1, r2 &\rightarrow p1, p3 \\
(p1) \text{ ld4}[r3] &\rightarrow r4 \\
\text{add } r4, r1 &\rightarrow r5 \text{ (no dependence if } p1=0) \\
\end{align*}
\]

- Defer stalls

Operand Delivery

- **Deferred Stall**
  - Stall actually in EXE stage
  - Clock frequency
  - Operand read over - can’t re-read
  - Snoop the register bypass network

- OLM - Operand Latch Manipulation
Execution

- **Deferred Stall**
- **Execute**
  - Writes turned off at retirement for false predicates
  - Different latencies - Out Of Order “Execution”
  - In-order retire - scoreboard

```plaintext
cmp.eq r1, r2 --> p1, p3
cmp.eq r7, r8 --> p5, p7
(p1) ld4[r3] --> r4 (reads p1 in EXE)
(p5) add r4, r1 --> r5 (reads p5 in REG)
```

- **Predicates**
  - ‘Producer’ reads in EXE
  - ‘Consumer’ reads in REG

Execution

- **Predicates**
  - Forward as soon as possible
  - Minimize forwarding logic
  - Predicate generation - deterministic latency

- Separate Register file
  - Speculative, Architectural (SPRF, APRF)
  - Shadow state
  - Bypass paths to eliminate ‘false stalls’
DET/WRB - Parallel Branches

- Multi-way branches - speculation + predication
- B units - up to 3 branches’ parallel execution
- Execution in DET stage
- Can use predicates in the same bundle

- Software pipeline support - LC, EC

DET/WRB - Parallel Branches

- Control Speculation
  - ld.s, chk.s
  - Exception Deferral - NaTs, NaTVals (poison bits!)
  - Store NaTs? - store.spill, ld.fill (context switch)
  - UNaT, RNaT

- Data Speculation
  - ld.a, chk.a, ld.c
  - ld.c can be issued with dependent instructions
  - ALAT - 32 entries, Register ID, Address, Size

- In-order retirement (branch misprediction/flush).
**FPU Details**

- Pipelined FMACs \((A \times B + C)\) (5 cycles)
- 4 DP ops/ 8 SP (SIMD) ops per cycle
- Divide/ Square root - S/W pipeline
- FP CMP operations (2 cycles)
- direct L2 cache contact - 2 ldf pair / cycle
- setf, getf, XMA, status registers

**Memory Subsystem**

- Address translation
  - 32 entry L1 DTLB, 96 entry L2 DTLB, Page size 4K - 256 M
  - Regions for sharing, Keys for protection
  - Hardware page walker
Memory Subsystem

- **L1 Data**
  - 16 K, 4-way, 32 byte lines
  - write through, no write allocate
  - dual ported, 2 cycle load latency

- **L2, on chip, unified**
  - 96 K, 6 way, 64 byte lines, Write back, write allocate
  - Dual ported, 6 cycles Int, 9 cycles FP load latencies
  - MESI protocol for coherence

- **L3, off chip, on package, unified**
  - 4 M, 4-way, 64 byte lines
  - 21-24 cycle latency, 128 bit bus

Memory Subsystem

- **Caches**
  - Hints
  - FP NT1 = Int NT2
  - Bias - Easier MESI

<table>
<thead>
<tr>
<th>Hint</th>
<th>Semantics</th>
<th>L1 response</th>
<th>L2 response</th>
<th>L3 response</th>
</tr>
</thead>
<tbody>
<tr>
<td>NT/A</td>
<td>Non-temporal (all levels)</td>
<td>Don't allocate</td>
<td>Allocate, mark as next replace</td>
<td>Don't allocate</td>
</tr>
<tr>
<td>NT/2</td>
<td>Non-temporal (2 levels)</td>
<td>Don't allocate</td>
<td>Allocate, mark as next replace</td>
<td>Normal allocation</td>
</tr>
<tr>
<td>NT/1</td>
<td>Non-temporal (1 level)</td>
<td>Don't allocate</td>
<td>Normal allocation</td>
<td>Normal allocation</td>
</tr>
<tr>
<td>T1 (default)</td>
<td>Temporal</td>
<td>Normal allocation</td>
<td>Normal allocation</td>
<td>Normal allocation</td>
</tr>
<tr>
<td>Bias</td>
<td>Initial to modify</td>
<td>Normal allocation</td>
<td>Allocate into exclusive state</td>
<td>Allocate into exclusive state</td>
</tr>
</tbody>
</table>
Rest of the Processor

- System Bus
  - 64 bit, 2.1GB/s,
  - Multidrop, Split transaction bus
  - Up to 56 outstanding transactions
  - Optimized MESI protocol
  - Glue-less multiprocessor support (Up to 4)

- IA 32 control

- ECC/Parity coverage of processor and bus
  - Read only structures - parity
  - Data - ECC.

Putting It All Together

The Block Diagram
Conclusions

To Sum Up
Conclusions

• Complexity shift to compilers
• Methods to express compile time information
• Large register files, EPIC specific Hardware
• Optimized FPUs for multimedia applications
• Large L3 cache
• Reliability and performance - server side

“Neat design, Let us see if it succeeds”