Introduction to Explicitly Parallel Instruction Computing (EPIC) and Very Long Instruction Word (VLIW) Architectures

Bhagi Narahari

Static ILP: VLIW/EPIC Architectures

• Overview of key Explicit Parallel Instruction Computing (EPIC) concepts
  – speculation, predication, register files
• Very Large Instruction Word (VLIW) and EPIC:
  – VLIW architectures progressed to EPIC
  – Let’s take a quick look at VLIW

VLIW and EPIC

• VLIW architectures progressed to EPIC
• A quick look at “pure” VLIW approach

VLIW: Very Large Instruction Word

• Each “instruction” has explicit coding for multiple operations
  – In IA-64, grouping called a “packet”
  – In Transmeta, grouping called a “molecule” (with “atoms” as ops)
• Tradeoff instruction space for simple decoding
  – The long instruction word has room for many operations
  – By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  – E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    • 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  – Need compiling technique that schedules across several branches
What Is VLIW?
- VLIW separately directs each functional unit

VLIW Instruction Execution

Historical Perspective: Microcoding, nanocoding (and RISC)

Principles of VLIW Operation
- Statically scheduled ILP architecture.
- Wide instructions specify many independent simple operations.
- Multiple functional units execute all of the operations in an instruction concurrently, providing fine-grain parallelism within each instruction.
- Instructions directly control the hardware with no interpretation and minimal decoding.
- A powerful optimizing compiler is responsible for locating and extracting ILP from the program and for scheduling operations to exploit the available parallel resources.

The processor does not make any run-time control decisions below the program level.

Realistic VLIW Datapath
Ideal Models for VLIW Machines

- Almost all VLIW research has been based upon an ideal processor model.
- This is primarily motivated by compiler algorithm developers to simplify scheduling algorithms and compiler data structures.
  - This model includes:
    - Multiple universal functional units
    - Single-cycle global register file
  and often:
    - Single-cycle execution
    - Unrestricted, Multi-ported memory
  and sometimes:
    - Unlimited resources (Functional units, registers, etc.)

Scheduling for Fine-Grain Parallelism

- The program is translated into primitive RISC-style (three address) operations
- Dataflow analysis is used to derive an operation precedence graph from a portion of the original program
- Operations which are independent can be scheduled to execute concurrently contingent upon the availability of resources
- The compiler manipulates the precedence graph through a variety of semantic-preserving transformations to expose additional parallelism

Example

The program is translated into three-address (3-address) operations:

Original Program:
- \( e = (a + b) \times (c + d) \)
- \( b++ \)

3-address Code:
- \( r1 = a + b \)
- \( r2 = c + d \)
- \( e = r1 \times r2 \)
- \( b = b + 1 \)

Dependency Graph:

- \( A \) and \( B \) are successors of \( D \).
- \( D \) is a predecessor of both \( A \) and \( B \).

VLIW List Scheduling – will return to List Scheduling later...

- Assign Priorities
- Compute Data Ready List - all operations whose predecessors have been scheduled.
- Select from DRL in priority order while checking resource constraints
- Add newly ready operations to DRL and repeat for next instruction

VLIW List Scheduling Example:

- 4-wide VLIW
- Data Ready List:
  - \( 1 \), \( 2 \), \( 3 \), \( 4 \), \( 5 \), \( 6 \)
  - \( 2 \), \( 3 \), \( 4 \), \( 5 \), \( 6 \)
  - \( 7 \), \( 8 \)
  - \( 9 \), \( 10 \), \( 11 \), \( 12 \)
  - \( 13 \)
Recall: Unrolled Loop that Minimizes Stalls for Scalar

1. Loop: L.D F0,0(R1)
2. L.D F6,-8(R1)  L.D to ADD.D: 1 Cycle
3. L.D F10,-16(R1) ADD.D to S.D: 2 Cycles
4. L.D F14,-24(R1)
5. ADD.D F4,F0,F2
6. ADD.D F8,F6,F2
7. ADD.D F12,F10,F2
8. ADD.D F16,F14,F2
9. S.D 0(R1),F4
10. S.D -8(R1),F8
11. S.D -16(R1),F12
12. DSUBUI R1,R1,#32
13. BNEZ R1,LOOP
14. S.D 8(R1),F16 ; 8-32 = -24

14 clock cycles, or 3.5 per iteration

Loop Unrolling in VLIW

<table>
<thead>
<tr>
<th>Memory</th>
<th>Memory</th>
<th>FP</th>
<th>FP</th>
<th>Int. op/ Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference 1</td>
<td>reference 2</td>
<td>operation 1</td>
<td>operation 2</td>
<td>branch</td>
</tr>
<tr>
<td>L.D F0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>L.D F14,-24(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>ADD.D F8,F6,F2</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F16,F14,F2</td>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>S.D 0(R1),F4</td>
<td>S.D -3(R1),F8</td>
<td>6</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>S.D -16(R1),F12</td>
<td>S.D -24(R1),F16</td>
<td>7</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>S.D -32(R1),F20</td>
<td>ADD.D F16,F14,F2</td>
<td>8</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>S.D -64(R1),F28</td>
<td>DSUBUI R1,R1,#48</td>
<td>9</td>
<td>0</td>
<td>9</td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
Average: 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW (15 vs. 6 in SS)

Enabling Technologies for VLIW

- VLIW Architectures achieve high performance through the combination of a number of key enabling hardware and software technologies.
  - Optimizing Schedulers (compilers)
  - Static Branch Prediction
  - Symbolic Memory Disambiguation
  - Predicated Execution
  - (Software) Speculative Execution
  - Program Compression

VLIW Design Issues

- Unresolved design issues
  - The best functional unit mix
  - Register file and interconnect topology
  - Memory system design
  - Best instruction format
- Many questions could be answered through experimental research
  - Difficult - needs effective retargetable compilers
- Compatibility issues still limit interest in general-purpose VLIW technology

However, VLIW may be the only way to build 8-16 operation/cycle machines.
VLIW vs. Superscalar

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Superscalar</th>
<th>VLIW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple instructions/cycle</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Multiple operations/instruction</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Instruction stream parsing</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Run-time analysis of register dependencies</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Run-time analysis of memory dependencies</td>
<td>maybe</td>
<td>occasionally</td>
</tr>
<tr>
<td>Runtime instruction reordering</td>
<td>maybe</td>
<td>no</td>
</tr>
<tr>
<td>Runtime register allocation</td>
<td>maybe (renaming)</td>
<td>maybe (iteration frames)</td>
</tr>
</tbody>
</table>

Real VLIW Machines

- VLIW Minisupercomputers/Superminicomputers:
  - Multiflow TRACE 7/300, 14/300, 28/300 [Josh Fisher]
  - Multiflow TRACE 600 [Bob Colwell]
  - Cydrome Cydra 5 [Bob Rau]
  - IBM Yorktown VLIW Computer (research machine)
- Single-Chip VLIW Processors:
  - Intel iWarp, Philip’s LIFE Chips (research)
- Single-Chip VLIW Media (through-put) Processors:
  - Trimedia, Chromatic, Micro-Unity
- DSP Processors (Texas Inst. TMS320C6x, Philips Trimedia)

Why VLIW Now?

- Technology Trends: Nonscalability of Superscalar Processor
  - ILP and complexity
- Better compilation technology

Performance Obstacles of Superscalars

- Branches
  - branch prediction helps, but penalty is still significant
  - limits scope of dynamic and static ILP analysis + code motion
- Memory Load Latency
  - CPU speed increases at 60% per year
  - memory speed increases only 5% per year
- Memory Dependence
  - disambiguation is hard, both in hardware and software
- Sequential Execution Semantics ISAs
  - total ordering of all the instructions
  - implicit inter-instruction dependences

Very expensive to implement wide dynamic superscalars
**Strengths of VLIW Technology**

- **Parallelism can be exploited at the instruction level**
  - Available in both vectorizable and sequential programs.
- **Hardware is regular and straightforward**
  - Most hardware is in the datapath performing useful computations.
  - Instruction issue costs scale approximately linearly.
  - Potentially very high clock rate.
- **Architecture is “Compiler Friendly”**
  - Implementation is completely exposed: 0 layer of interpretation.
  - Compile time information is easily propagated to run time.
- **Exceptions and interrupts are easily managed**
- **Run-time behavior is highly predictable**
  - Allows real-time applications.
  - Greater potential for code optimization.

---

**Weaknesses of VLIW Technology**

- **No object code compatibility between generations**
- **Program size is large (explicit NOPs)**
  - Multiflow machines predated “dynamic memory compression” by encoding NOPs in the instruction memory.
- **Compilers are extremely complex**
  - Assembly code is almost impossible.
- **Philosophically incompatible with caching techniques**
- **VLIW memory systems can be very complex**
  - Simple memory systems may provide very low performance.
  - Program controlled multi-layer, multi-banked memory.
- **Parallelism is underutilized for some algorithms.**

---

**The EPIC Model**

- **VLIW concept in terms of static ILP**
  - Use compiler to extract parallelism.
- **Try to overcome limitations of VLIW**
- **Can we use additional H/W support to enhance S/W techniques?**

---

**EPIC Concepts**

- **Explicitly Parallel Instruction Computing**
  - Unlike early VLIW designs, EPIC does not use fixed width instructions...as many parallel as possible!
- **Programs must be written using sequential semantics**
  - Parallel semantics not supported.
  - Explicitly lay out the parallelism.
  - Eg: swapping of operands.
**Intel/HP EPIC/IA-64 Architecture**

- **EPIC (Explicitly Parallel Instruction Computing)**
  - An ISA philosophy/approach
  - Very closely related to but not the same as VLIW

- **IA-64**
  - An ISA definition
  - e.g. IA-32 (was called x86), PA-RISC
  - Intel’s new 64-bit ISA
  - An EPIC type ISA

- **Itanium (was code named Merced)**
  - A processor implementation of an ISA
  - The first implementation of the IA-64 ISA

**IA-64 EPIC vs. Classic VLIW**

- **Similarities:**
  - Compiler generated wide instructions
  - Static detection of dependencies
  - ILP encoded in the binary (a group)
  - Large number of architected registers

- **Differences:**
  - Instructions in a bundle can have dependencies
  - Hardware interlock between dependent instructions
  - Accommodates varying number of functional units and latencies
  - Allows dynamic scheduling and functional unit binding
  - Static scheduling are “suggestive” rather than absolute
  - Code compatibility across generations

  ⇒ Code compatibility across generations

  but software won’t run at top speed until it is recompiled so

  “shrink-wrap binary” might need to include multiple builds

**EPIC: Key Concepts**

- Speculation
- Predication (and parallel compares)
- Large (Rotating) Register Files

**EPIC Concepts: Speculation**

- What do you do with all the parallelism and how
  - traditional problem has been that we never have enough work ready in order to keep a machine fully busy

- what happens when you stop worrying about only doing things we must
  - if we have the power of parallelism, key is to not throw it away
  - anytime processor is ready to do six things, do not give it only two things to do and ignore ability to do more
  - how?
### EPIC Concepts: Speculation

- Speculatively ask machine to do more things
- Pick tasks that might be needed in future
  - Just aren’t sure whether they will be needed at the time
  - Make sure you can determine if they will be needed
  - Extra tasks do not involve time (due to parallel units)
  - Even if they useful only 50% of the time, we have completed 50% of the tasks ahead of time!
- Promise of EPIC based on speculation
  - Goal is to compute things before they are needed, so when program needs result it is already there!
- Note: Branch Prediction is part of speculation but not all of speculation concept

### EPIC Concepts: Predication

- Branching is generally bad because it interferes with the ideal pipeline model of reading instructions while earlier inst is executed
- Ideally, if we eliminate branches then this problem disappears
  - Can we lin instruction to a condition?
- Predication is process by which branches are eliminated
  - Note: predication is not branch prediction!!

### EPIC Concepts: Predication

- Predication allows instructions to execute in parallel, with some “on” and some “off” but without need for branches
  - Every instruction written with a specified predicate register to control whether instruction executes at runtime
- Ability to do “parallel compares”
  - Ability to compute and combine comparison operations in parallel
  - $(A>B)$ and $(B<0)$ can be computed in parallel using parallel compare instructions

### EPIC Concepts: Predication

- EPIC provides predicated instructions
  - Every instruction can be executed in predicated manner
  - Instruction execution tied to result of a predicate register
  - One predicate register hardwired to a 1; use this to always execute
**EPIC Concepts: Predication**

If \((a > b)\) {
    \(x = a\)
    \(z = 1\)
} else {
    \(x = b\)
    \(z = z + 1\)
}

Test = TRUE if \((a > b)\), else FALSE
if (test is TRUE) \(tmp1 = a\)
if (test is FALSE) \(tmp1 = b\)
\(x = tmp1\)
if (test is TRUE) \(tmp2 = 1\)
if (test is FALSE) \(tmp2 = z + 1\)
\(z = tmp2\)

note: No branches above!

---

**Predication**

```
pr1 ← cmp ...
...
be X if pr1 ...
```

```
pr1 ← cmp ...
r1 ← r2 - r3 if pr1 ...
r1 ← r2 + r3 if !pr1
```

---

**Predicated Execution**

- Each instruction can be separately predicated
- 64 one-bit predicate registers in IA-64
  - An instruction is effectively a NOP if its predicate is false
  - Converts control flow into dataflow

---

**CS 211**
**How does Predication help ILP?**

- Two paths of branch become independent instructions – tagged by predicate register
- Can send both into execution pipeline
  - Parallel execution
- Will retire/commit instruction when predicate register is set – if not set then effectively a NOP
  - Note similarity to use of speculative tags in superscalar

**Compare Operations**

- Comparison operations may
  - set predicate registers (up to two simultaneously)
  - compare to a GPR
- Comparison operations themselves can be predicated
- Predicate registers may be combined by logical operations

---

**If-conversion for Predication**

- Identifying region of basic blocks based on resource requirement and profitability (branch mis-prediction rate, mis-prediction cost, and parallelism)
- Result: a predicated block

**Reducing Control Height with parallel compares**

- Convert nested if’s into a single predicate
- Result: shorter control path by reducing the number of branches
Multiway Branch Example

- Use Multiway branches
  - Speculate compare (i.e. move above branch)
  - Do not reduce number of branches

```
cmp.lt p1,p0=a,b
(p1) br X;
cmp.lt p2,p0=b,c
(p2) br Z;;
```

Predication is not Control Speculation

- Two type of speculation:
  - data speculation
  - control speculation

- In data speculation: loads are moved ahead of stores (will be discussed later)

- In control speculation: instructions are moved from below a branch to above a branch
  - control speculation ≠ predication

Differences

- In predication:
  - compare instruction sets predicate registers
  - predicate registers used to confirm instructions and annul others
  - exceptions take place where they occur

```
cmp.lt p1,p0=a,b
(p1) br X;
```

Differences

- In control speculation:
  - instructions below a branch is moved to above it and marked as speculative
  - exceptions are postponed via register tagging
  - if speculation turns out to be false, result is discarded (register overwritten)
  - if speculation turns out to be true, must check whether speculative instructions caused exceptions

```
cmp.lt p1,p0=a,b
(p1) br X
(p2) br Z;;
```
CS 211

Control Speculation

Note: If the speculation caused a delayed exception, then r1's speculation tag is set. However, in X, where the speculation is unwanted, r1 is simply overwritten - r2 and r3 being non-speculative will cause the speculation tag of r1 to be cleared.

Note: In the block where speculation is wanted, r1 is a source. A source of a non-speculative instruction with a speculation tag set will cause an exception!

CS 211

Control and Data Speculation

- Two kinds of instructions in IA-64 programs
  - Non-speculative instructions -- known to be useful/needed
    - Would have been executed in the original program
  - Speculative instructions -- may or may not be used
    - Schedule operations before results are known to be needed
    - Usually boosts performance, but occasionally may degrade
    - Heuristics can guide compiler in aggressiveness

- Two kinds of speculation
  - Control and Data

- Moving loads up is a key to performance
  - Hide increasing memory latency
  - Computation chains frequently begin with loads
Architectural Support for Control Speculation

- 65th bit (NaT bit) on each GR indicates if an exception has occurred
  - Plus which part of code it went to
- Special speculative loads that set the NaT bit if a deferrable exception occurs
- Special chk.s instruction that checks the NaT bit and branches to recovery, if set
- Computational instructions propagate NaTs like IEEE NaN's
- Compare operations propagate “false” when writing predicates

Architectural Support for Data Speculation

- ALAT – Advanced Load Architecture Table: HW structure containing information about outstanding loads advanced across stores
- Instructions
  - ld.a - advanced loads
  - ld.c - check loads
  - chk.a - advance load checks
  - aliasing st invalidating entries in ALAT
- Speculative advanced loads - ld.sa - is a control speculative advanced load with fault deferral (combines ld.a and ld.s)

Speculative, Non-Faulting Load

- ld.s fetches speculatively from memory
  i.e. any exception due to ld.s is suppressed
- If ld.s did not cause an exception then chk.s r is an NOP, else a branch is taken (to some compensation code)

Speculative, Non-Faulting Load

- Speculatively load data can be consumed prior to check
- “speculation” status is propagated with speculated data
- Any instruction that uses a speculative result also becomes speculative itself (i.e. suppressed exceptions)
- chk.s checks the entire dataflow sequence for exceptions
**Advanced Load**

- What if we have a load unit free?
  - Even within a basic block?
- Move a load instruction to an earlier slot: what do we have to watch out for??
  - Load R1, (R2) assume R2=100
- Make sure no intervening store has written to same address
  - Has store has written anything into address 100?
    - Yes= redo load again, No= use the loaded data

**“Advanced” Load**

- ld.a starts the monitoring of any store to the same address as the advanced load
- If no aliasing has occurred since ld.a, ld.c is a NOP
- If aliasing has occurred, ld.c re-loads from memory

**Using Load Results**

- inst 1
  - inst 2
  - st [?] 
  - ld r1=[x]
    - use=r1
- ld a r1=[x]
  - inst 1
  - inst 2
  - use=r1
  - st [?] 
  - ld c r1=[x]
    - use=r1

**EPIC Concepts: Rotating Register Sets and Software Pipelining**

- Speculation and Predication require large sets of registers in EPIC
- In addition, concept of Rotating Register Sets to support Software Pipelining
  - to help with execution of loops
  - extend pipeline concept
- Recall Loop unrolling concept:
  - Unroll loop iterations: perform multiple iterations concurrently
  - Can explode the code size
Software Pipelining: also known as Modulo Scheduling

Conventional Sequential Execution of iterations

Loop Body

Overlapped Execution by Pipelining iterations

Less time overall

Software Pipelining Example

Four Independent Instructions

Software Pipeline

New Loop Body

ILP = 4

Epilogue

Rotating Loop Frames for Loop Pipelining

Suppose B is only data dependent (through data stored in registers) on A, and C only on B.
- The “pipelined” kernel block (containing independent computation from C, B, and A) potentially has better ILP.

What happens if C is also data dependent on A.
- The result placed in register by A gets clobbered by the next execution of A (in the next cycle) before C can use it two cycles from now.

Nice Loop Pipelining Example

i=0
while (i<99) {
    Rx = a[i]
    Ry = Rx / 10
    a[i] = Ry
    Rx = a[i+1]
    Ry = Rx / 10
    a[i+1] = Ry
    Rx = a[i+2]
    Ry = Rx / 10
    a[i+2] = Ry
    i++
}

i=0
while (i<99) {
    Rx = a[i]
    Ry = Rx / 10
    a[i] = Ry
    Rx = a[i+1]
    Ry = Rx / 10
    a[i+1] = Ry
    Rx = a[i+2]
    Ry = Rx / 10
    a[i+2] = Ry
    i++
}

i=0
while (i<99) {
    Rx = a[i]
    Ry = Rx / 10
    a[i] = Ry
    Rx = a[i+1]
    Ry = Rx / 10
    a[i+1] = Ry
    Rx = a[i+2]
    Ry = Rx / 10
    a[i+2] = Ry
    i++
}

i=0
while (i<99) {
    Rx = a[i]
    Ry = Rx / 10
    a[i] = Ry
    Rx = a[i+1]
    Ry = Rx / 10
    a[i+1] = Ry
    Rx = a[i+2]
    Ry = Rx / 10
    a[i+2] = Ry
    i++
}

i=0
while (i<99) {
    Rx = a[i]
    Ry = Rx / 10
    a[i] = Ry
    Rx = a[i+1]
    Ry = Rx / 10
    a[i+1] = Ry
    Rx = a[i+2]
    Ry = Rx / 10
    a[i+2] = Ry
    i++
}

i=0
while (i<99) {
    Rx = a[i]
    Ry = Rx / 10
    a[i] = Ry
    Rx = a[i+1]
    Ry = Rx / 10
    a[i+1] = Ry
    Rx = a[i+2]
    Ry = Rx / 10
    a[i+2] = Ry
    i++
}

i=0
while (i<99) {
    Rx = a[i]
    Ry = Rx / 10
    a[i] = Ry
    Rx = a[i+1]
    Ry = Rx / 10
    a[i+1] = Ry
    Rx = a[i+2]
    Ry = Rx / 10
    a[i+2] = Ry
    i++
}
EPIC Concepts: Software Pipelining

- Software pipeline (also known as Modulo Scheduling) requires ‘register reuse’
  - instead provide Rotating Register Sets
- In IA-64 Registers organized into set of 32 static registers and 96 rotating registers
  - R[1], R[2]...R[32]
  - RRB: Rotating register base
    - Can reference one set of rotating registers in an iteration, and the HW takes care of assigning others
    - can allow same references in four different iterations

Register Files Static/Rotating

- Each register file may have a static and a rotating portion
- The i^th rotating register in file F is named F[i]
  - F[i] = Fr [iRRB + i) % size(Fr)]

Rotating Registers: Example

- Let loop body use: r1,r2,r3,r4
- Can specify RRB (base) = 33
  - First iteration will get 33, 34, 35,36
  - Second iteration will get 37,38,39,40
  - i.e., modulo 4

Register Renaming

- 128 general purpose physical integer registers
- Register names R0 to R31 are static and refer to the first 32 physical GPRs
- Register names R32 to R127 are known as “rotating registers” and are renamed onto the remaining 96 physical registers by an offset
- Remapping wraps around the rotating registers such that when offset is non-zero, physical location of R127 is just below R32
Renaming with Rotating Registers

i = 0
Ry = a[0] / 10
Rx = a[1]

while (i < 97) {
    a[i] = Ry * Rx'
    Ry = Rx / 10
    Rx' = Rx
    Rx = a[i + 2]
    i++
}
a[97] = Ry + Rx'
a[98] = Rx / 10 + Rx

EPIC Summary

- H/W techniques added to pure VLIW S/W techniques
  - Speculation support
  - Predicated execution model
  - Rotating registers for s/w pipelining
- Next:
  - An example EPIC: IA-64
    - How to solve the code compatibility issue?
    - How to deal with recompiling when changing the microarchitecture?
    - Concept of Bundle Scheduling
  - Some advanced superscalar features: branch prediction

Loop Pipelining Requiring Renaming

i = 0
while (i < 99) {
    a[i] = a[i] / 10 + a[i]
    Rx = a[i]
    Ry = Rx / 10
    a[i] = Ry + Rx
    i++
}
a[99] = Ry + Rx'
a[100] = Rx / 10 + Rx

Quick Look at an EPIC Processor: The IA-64 Architecture

- 128 general-purpose registers
- 128 floating-point registers
- Arbitrary number of functional units
- Arbitrary latencies on the functional units
- Arbitrary number of memory ports
- Arbitrary implementation of the memory hierarchy

Needs re-targetable compiler and recompilation to achieve maximum program performance on different IA-64 implementations
IA-64 Instruction Format

- **IA-64 “Bundle”**
  - Total of 128 bits
  - Contains three IA-64 instructions (aka syllables)
  - Template bits in each bundle specify dependencies both within a bundle as well as between sequential bundles
  - A collection of independent bundles forms a “group”
    A more efficient and flexible way to encode ILP then a fixed VLIW format

- **IA-64 Instruction**
  - Fixed-length 40 bits long
  - Contains three 7-bit register specifiers
  - Contains a 6-bit field for specifying one of the 64 one-bit predicate registers

---

IA-64: Groups & Bundles

- **A group is a set of “parallel”/independent instructions**
  - All operations in a group can be executed in parallel if there are no resource limits
  - A STOP command indicates end of group

- **A bundle is what is presented to a processor**
  - Group consists of a number of bundles
  - A bundle sent to the processor

- **What is the advantage of this?**

---

Cool Features of IA64

- Predicated execution
- Speculative, non-faulting Load instruction
- Software-assisted branch prediction
- Register stack
- Rotating register frame
- Software-assisted memory hierarchy

*Mostly adapted from mechanisms that had existed for VLIWs*

---

Itanium Specifics

- 6-wide 10-stage pipeline
- Fetch 2 bundles per cycle with the help of BP into a 8-bundle deep fetch queue
- 512-entry 2-level BPT, 64-entry BTAC, 4 TAR, and a RSB
- Issue up to 2 bundles per cycle some mixes of 6 instructions e.g. (MFL,MFI) or (MIB,MIBh)
- Can issue as little as one syllable per cycle on RAW hazard interlock or structural hazard (scoreboard for RAW detection)
- 8R-6W 128 Entry Int. GPR, 128 82-bit FPR, 64 predicate reg’s
- 4 globally-bypassed single-cycle integer ALUs with MMX, 2 FMACs, 2 LSUs, 3 BUs
- Can execute IA-32 software directly

*Intended for high-end server and workstations
You can buy one now, finally.*
IA-64 EPIC vs. Classic VLIW

• **Similarities:**
  – Compiler generated wide instructions
  – Static detection of dependencies
  – ILP encoded in the binary (a group)
  – Large number of architected registers

• **Differences:**
  – Instructions in a bundle can have dependencies
  – Hardware interlock between dependent instructions
  – Accommodates varying number of functional units and latencies
  – Allows dynamic scheduling and functional unit binding
    
    Static scheduling are "suggestive" rather than absolute

⇒ Code compatibility across generations
  
  but software won’t run at top speed until it is recompiled so "shrink-wrap binary" might need to include multiple builds

---

EPIC and Compiler Optimization

• EPIC requires dependency free “scheduled code”
• Burden of extracting parallelism falls on compiler
• success of EPIC architectures depends on efficiency of Compilers!!
• We provide overview of Compiler Optimization techniques (as they apply to EPIC/ILP)