Microprocessors

Introduction to ia64 Architecture
Jan 31st, 2002
General Principles
Instruction Level Parallelism

Certain instructions can be executed in parallel

Certain instructions can be executed in any order

Both of these stem from lack of dependency between instructions.

The goal of the ia64 design:

- Exploit ILP more effectively
Explicitly Parallel Instruction Computing

- Conventional RISC
  - Processor discovers and exploits ILP
- Conventional VLIW
  - Programmer knows the precise execution model and explicitly lays out the program to take advantage of ILP
- EPIC
  - Programmer indicates possible ILP, processor does the rest of the job.
The ia64 Architecture

Instructions are bundled in packets of 3

- Packet length is 128 bits
- Three 41-bit instructions in each packet
- 5-bits of scheduling information

Scheduling information indicates

- What functional units required for each instruction in the packet.
- What instructions can be executed in parallel
Instruction Bundles

An instruction bundle is a group of instructions that can be executed in parallel:

- **No read after write dependencies**
  - That’s where one instruction writes a value to memory or a register that is read by another instruction.

- **No write after write dependencies**
  - That’s where two instructions write the same register or location in memory.
More on Bundles

The scheduling bits indicate the length of a particular instruction bundle

- At one extreme, one instruction per bundle, no parallelism, works but slow!
- At the other extreme, can join packets together to make bundles of arbitrary length
- Compiler is supposed to construct bundles as big as possible, but does not otherwise have to worry about latencies for correctness.
Bundles and MP Versions

Versions of the ia64 implementation may differ in their capabilities of executing instructions in parallel.

If a bundle is larger than what the implementation can handle, it just breaks it up into pieces done sequentially.

Unlike VLIW, or even conventional RISC, no need to recompile for new versions of processors.
Bundles and Jumps

A jump can dynamically end a bundle
First jump to take ends bundle dynamically
So it is permissible to have multiple jumps in one bundle. Processor takes care of this.
The compiler needs to do an analysis to find ILP to construct the largest possible bundles.

In some cases, this may entail predication, trace scheduling, speculative execution etc. These can all be done as much as the compiler wants, but are not required.
Speculative Execution, Predication

All instructions are predicated
Large number of predicate registers
Instruction effective only if predicated
Allows larger bundles

- For example, can have all instructions of both the then and else branches of an IF statement in a single bundle with only the relevant branch being actually executed
If instructions are executed speculatively, i.e. you don’t know if they should be executed or not, some instruction may give a garbage value (e.g. divide by zero). Don’t want a trap, since perhaps we will find out in a moment that we should discard the whole thread. Therefore, must silently propagate indication of bad value (not a value).
Speculative Execution, Loads

Loads can cause pipeline stalls
Therefore you want to do them early
But danger in moving them across stores
So there is a load-predict instruction

- Please load this value, I think I will need it

And a load confirm instruction

- OK, now I want that value, check no one stored there since my load predict. If so, too bad you will have to go load it now.
Lots and Lots of Registers

The ia64 has hundreds of user level registers.

Easier to do speculative execution in registers.

As usual, we hate loads, so avoid them.

Instructions not limited to 32 bits, so we can afford long register identifier fields.
Register Windows

Register windows are provided

- Like the SPARC, except that you can say how much to move the window by
- Overlap between caller and callee possible as on the SPARC
- But if you only need a few registers you don't need to consume a large fixed chunk of registers.
- (old idea, AMD29K had a similar design)
Efficient Code for Loops

Suppose we have a loop whose form is:

- Load value
- Add some constant to that value
- Store result

That’s nasty for dependencies:

- We want space between the load and the add
- And space between the add and the store
Loop Unrolling and Software Pipelining

If we unroll several iterations of the loop we can be doing an add of previous iteration while loading the next.

Generates much more code.

Requires complex prolog (get things started) and epilog (finish things off) code.

In practice, hard to apply in all cases.
Rotating Registers

Suppose we generate code for the loop

- Load register R7 with input value
- Add constant to register R8
- Store register R9 to memory

Certainly no dependencies

- But code looks wrong and useless!

How can we make the above make sense?
More on Rotating Registers

Here is the code

- Load register R7 with input value
- Add constant to register R8
- Store register R9 to memory

Now renumber registers on each loop

- Old R7 is new R8
- Old R8 is new R9
- Old R9 is new R7

Ah ha! Magic, the generated code is OK!
More on Rotating Registers

Limited subsets of registers can rotate:
- Giving the renumbering on previous slide
- The loop instruction automatically triggers the rotation (a bit like registers windows)
- Special prolog/epilog counts deal with setup and cleanup cases

Voila! Efficient loops without
- Loop unrolling
- Software pipelining
The Bottom Line

The advantages of VLIW
- Greater ILP exploitation
- Simpler hardware

Without the disadvantages
- Code does not depend on processor model

But
- We still depend on the compiler a whole lot!

Next time: Details of the ia64 architecture