Today

- Software Pipelining
- EPIC
- IA-64

Time Permitting
- Binary Translation
Software Pipelining

• For (int i=0; i<n; i++)

Example: Machine Instructions

• For (int i=0; i<n; i++)

• t1 = i-1
• t2 = t1 << 2;
• t3 = A + t2
• t4 = i << 2
• t5 = A + t4
• t6 = *t3
• t7 = i*i
• t8 = t6 + t7
• *t5 = t8
• i = i + 1
• t9 = n - i
• Bpos t9, loop
Example: ILP

• For (int i=0; i<n; i++)

• t1=i-1, t4=i<<2, t7=i*i, i=i+1, t9=n-i
• t2=t1<<2, t5=A+t4, Bpos t9, loop
• t3=A+t2
• t6=*t3
• t8=t6+t7
• *t5=t8

Example

• t1=i-1, t4=i<<2, t7=i*i, i=i+1, t9=n-i
• t2=t1<<2, t5=A+t4, Bpos t9, loop
• t3=A+t2
• t6=*t3
• t8=t6+t7
• *t5=t8
Example

- $t_1=i-1$, $t_4=i<<2$, $t_7=i*i$, $i=i+1$, $t_9=n-i$
- $t_2=t_1<<2$, $t_5=A+t_4$, Bpos $t_9$, loop
- $t_3=A+t_2$
- $t_6=t_3$
- $t_8=t_6+t_7$
- $t_9=n-i$
- $t_5=A+t_4$, Bpos $t_9$, loop
- $t_8=t_6+t_7$
- $t_5=t_8$

Pipeline Loop body

Example: Software Pipeline

- $t_1=i-1$, $t_4=i<<2$, $t_7=i*i$, $i=i+1$, $t_9=n-i$
- $t_2=t_1<<2$, $t_5=A+t_4$, Bpos $t_9$, end1
- $t_3=A+t_2$
- $t_6=t_3$
- $t_8=t_6+t_7$
- $t_5=t_8$
- $t_9=n-1$
- $t_6=t_3$, $t_2=t_1<<2$, $t_5=A+t_4$, Bpos $t_9$, end2
- $t_7b=t_7$
- $t_5=t_8$
- $t_8=t_6+t_7$
- $t_6=t_3$
- $t_8=t_6+t_7$
- $t_5=t_8$
- $t_9=n-1$
- $t_5=t_8$
- $t_6=t_3$
- $t_8=t_6+t_7$
- $t_5=t_8$
EPIC

Scaling Idea

• Problem:
  – VLIW: amount of parallelism fixed by VLIW schedule
  – SuperScalar: have to check many dynamic dependencies

• Idealized Solution:
  – expose all the parallelism you can
  – run it as sequential/parallel as necessary
Basic Idea

• What if we scheduled an infinitely wide VLIW?

• For an N-issue machine
  – for I = 1 to (width of this instruction/N)
    • grab next N instructions and issue

Problems?

• Instructions arbitrarily long?
• Need infinite registers to support infinite parallelism?
• Split Register file still work?
• Sequentializing semantically parallel operations introduce hazards?
Instruction Length

- Field in standard way
  - \textit{pinsts} (from cs184a)
  - like RISC instruction components
- Allow variable fields (syllables) per parallel component
- Encode
  - stop bit (break between instructions)
  - (could have been length...)

Registers

- Compromise on fixed number of registers
  - ...will limit parallelism, and hence scalability...
- Also keep(adopt) monolithic/global register file
  - syllables can’t control which “cluster” in which they’ll run
  - \textit{E.g.} consider series of 7 syllable ops
    - where do syllables end up on 3-issue, 4-issue machine?
Sequentializing Parallel

• Consider wide instruction:
  – MUL R1, R2, R3   ADD R2, R1, R5

• Now sequentialize:
  – MUL R1, R2, R3
  – ADD R2, R1, R5

• Different semantics

Semantics of a “Long Instruction”

• Correct if executed in parallel
• Preserved with sequentialization
• So:
  – read values are from beginning of issue group
  – no RAW hazards:
    • can’t write to a register used as a source
  – no WAW hazards:
    • can’t write to a register multiple times
Non-VLIW-ness

Register File

- Monolithic register file
- Ports grows with number of physical syllables supported
Bypass

- VLIW
  - schedule around delay cycles in pipe
- EPIC not know which instructions in pipe at compile time
  - do have to watch for hazards between instruction groups
  - ? Similar pipelining issues to RISC/superscalar?
  - Bypass only at issue group boundary
    - maybe can afford to be more spartan?

Concrete Details

(IA-64)
Terminology

• Syllables (their pinsts)
• bundles: group of 3 syllables for IA-64
• Instruction group: “variable length” issue set
  – *i.e.* set of bundles (syllables) which may execute in parallel

IA-64 Encoding

Figure 3-16. Bundle Format

<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>1-unit or M-unit</td>
</tr>
<tr>
<td>I</td>
<td>Non-ALU integer</td>
<td>1-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>1-unit</td>
</tr>
</tbody>
</table>

Source: Intel/HP IA-64 Application ISA Guide 1.0
IA-64 Templates

Table 3-4: Template Field Encoding and Instruction Slot Mapping

<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</td>
</tr>
</tbody>
</table>

Source: Intel/HP IA-64 Application ISA Guide 1.0

IA-64 Registers

Figure 3-1. Application Register Model

Source: Intel/HP IA-64 Application ISA Guide 1.0
Other Additions

Other Stuff

- Speculation/Exceptions
- Predication
- Branching
- Memory
- Register Renaming
Speculation

- Can mark instructions as speculative
- Bogus results turn into designated NaT
  - (NaT = Not a Thing)
  - particularly loads
    - compare position bits
- NaT arithmetic produces NaTs
- Check for NaTs if/when care about result

Predication

- Already seen conditional moves
- Almost every operation here is conditional
  - (similar to ARM?)
- Full set of predicate registers
  - few instructions for calculating composite predicates
- Again, exploit parallelism and avoid losing trace on small, unpredictable branches
  - can be better to do both than branch wrong
Predication: Quantification

Predication Increases Performance

Branching

• Unpack branch
  – branch prepare (calculate target)
    • added branch registers for
  – compare (will I branch?)
  – branch execute (transfer control now)
• sequential semantics w/in instruction group
• indicate static or dynamic branch predict
• loop instruction (fixed trip loops)
• multiway branch (with predicates)
Memory

- **Prefetch**
  - typically non-binding?
- **control caching**
  - can specify not to allocate in cache
    - if know use once
    - suspect no temporal locality
  - can specify appropriate cache level
- **speculation**

Memory Speculation

- Ordering limits due to aliasing
  - don’t know if can reorder a[i], a[j]
    - a[j]=x+y;
    - C=a[i]*Z;
  - might get WAR hazards
- **Memory speculation**:
  - reorder read
  - check in order and correct if incorrect
  - Extension of VLIW common case fast / off-trace patchup philosophy
Memory Speculation

- store(st_addr, data)
- load(ld_addr, target)
- use(target)

- aload(ld_addr, target)
- store(st_adder, data)
- acheck(target, recovery_addr)
- use(target)

If advanced load fails, checking load performs actual load.
Memory Speculation

<table>
<thead>
<tr>
<th>Before Data Speculation</th>
<th>After Data Speculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>// other instructions</td>
<td></td>
</tr>
<tr>
<td>st8 [r4] = r12</td>
<td></td>
</tr>
<tr>
<td>ld8 r6 = [r8];</td>
<td></td>
</tr>
<tr>
<td>add r5 = r6, r7;</td>
<td></td>
</tr>
<tr>
<td>st8 [r18] = r5</td>
<td></td>
</tr>
<tr>
<td>ld8.a r6 = [r8];</td>
<td>// other instructions</td>
</tr>
<tr>
<td></td>
<td>add r5 = r6, r7;</td>
</tr>
<tr>
<td></td>
<td>// other instructions</td>
</tr>
<tr>
<td></td>
<td>st8 [r4] = r12</td>
</tr>
<tr>
<td></td>
<td>chk.a.clr r6, recover</td>
</tr>
<tr>
<td></td>
<td>back</td>
</tr>
<tr>
<td></td>
<td>st8 [r18] = r5</td>
</tr>
<tr>
<td></td>
<td>// somewhere else in program</td>
</tr>
<tr>
<td></td>
<td>recover</td>
</tr>
<tr>
<td></td>
<td>ld8 r6 = [r8];</td>
</tr>
<tr>
<td></td>
<td>add r5 = r6, r7</td>
</tr>
<tr>
<td></td>
<td>br back</td>
</tr>
</tbody>
</table>

If advanced load succeeds, values are good and can continue; otherwise have to execute patch up code.

Advanced Load Support

- Advanced Load Table
- Speculative loads allocate space in ALAT
  - tagged by target register
- ALAT checked against stores
  - invalidated if see overwrite
- At check or load
  - if find valid entry, advanced load succeeded
  - if not find entry, failed
    - reload ...or...
    - branch to patchup code
Register “renaming”

• Use top 96 registers like a stack?
• Still register addressable
• But increment base on
  – loops, procedure entry
• Treated like stack with “automatic”
  background task to save/restore values

Register “renaming”

• Application benefits:
  – software pipelining without unrolling
  – values from previous iterations of loop get
different names (rename all registers
allocated in loop by incrementing base)
  • allows reference to by different names
  – pass data through registers
    • without compiling caller/callee together
    • variable number of registers
Register “Renaming”

• …old bad idea?
  – Stack machines?
    • Does allow register named access
  – Register Windows (RISC-II, SPARC)
    • SPARC register windows were fixed size
    • had to save and restore in that sized chunk
    • only window-set visible

Register “renaming” Costs

• Slow down register access
  – have to do arithmetic on register numbers
• Require hardware register save/restore engine
  – orthogonal task to execution
  – complicated?
• Complicates architecture
Some Data (Integer Programs)

H&P Fig. 4.16 e3

Some Data (FP Programs)

H&P Fig. 4.17 e3
Binary Translation

Problem

• Lifetime of programs >> lifetime of piece of hardware (technology generation)
• Getting high performance out of old, binary code in hardware is expensive
  – superscalar overhead…
• Recompilation not viable
  – only ABI seems well enough defined; captures and encapsulates whole program
• There are newer/better architectures that can exploit hardware parallelism
Idea

• Treat ABI as a source language
  – the specification
• Cross compile (translate) old ISA to new architecture (ISA?)
• Do it below the model level
  – user doesn’t need to be cognizant of translation
• Run on simpler/cheaper/faster/newer hardware

Complications

• User visibility
• Preserving semantics
  – e.g. condition code generation
• Interfacing
  – preserve visible machine state
  – interrupt state
• Finding the code
  – self-modifying/runtime generated code
  – library code
Base

• Each operation has a meaning
  – behavior
  – affect on state of machine
• stws r29, 8(r8)
  – tmp=r8+8
  – store r29 into [tmp]
• add r1,r2,r3
  – r1=(r2+r3) mod 2^{31}
  – carry flag = (r2+r3>= 2^{31})

Capture Meaning

• Build flowgraph of instruction semantics
  – not unlike the IR (intermediate representation) for a compiler
    • what use to translate from a high-level language to ISA/machine code
  – e.g. IR saw for Bulldog (trace scheduling)
Optimize

- Use IR/flowgraph
  - eliminate dead code
    - esp. dead conditionals
    - e.g. carry set which is not used
  - figure out scheduling flexibility
    - find ILP

Trace Schedule

- Reorganize code
- Pick traces as linearize
- Cover with target machine operations
- Allocate registers
  - (rename registers)
  - may have to preserve register assignments at some boundaries
- Write out code
Details

- Seldom instruction→instruction transliteration
  - extra semantics (condition codes)
  - multi-instruction sequences
    - loading large constants
    - procedure call return
  - different power
    - offset addressing?,
    - compare and branch vs. branch on register
- Often want to recognize code sequence

Complications

- How do we find the code?
  - Known starting point
  - ? Entry points
  - walk the code
  - …but, ultimately, executing the code is the original semantic definition
    - may not exist until branch to...
Finding the Code

• **Problem:** can’t always identify statically
• **Solution:** wait until “execution” finds it
  – delayed binding
  – when branch to a segment of code,
    • certainly know where it is
    • and need to run it
  – translate code when branch to it
    • first time
    • nth-time?

Binary Translation

• *(finish up next lecture)*
Big Ideas [EPIC]

- Compile for maximum parallelism
- Sequentialize as necessary
  - (moderately) cheap

Big Ideas [IA-64 1]

- Latency reduction hard
  - path length is our parallelism limiter
  - often good to trade more work for shorter critical path
    - area-time tradeoff
  - speculation, predication reduce path length
    - perhaps at cost of more total operations
Big Ideas [IA64 2]

- Local control (predication)
  - costs issue
  - increases predictability, parallelism
- Common Case/Speculation
  - avoid worst-case pessimism on memory operations
  - common case faster
  - correct in all cases

Big Ideas [Binary Trans]

- Well-defined model
  - High value for longevity
  - Preserve semantics of model
  - How implemented irrelevant
- Hoist work to earliest possible binding time
  - dependencies, parallelism, renaming
  - hoist ahead of execution
    - ahead of heavy use
    - reuse work across many uses
- Use feedback to discover common case