Instruction Level Parallelism
- static instruction scheduling, software pipelining, VLIW, EPIC, and multi-threading

February 2018
Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson’s *Computer Architecture, a quantitative approach* (3rd and 4th eds), and on the lecture slides of David Patterson and John Kubiatowicz’s Berkeley course.
Overview

Up to now: Dynamic scheduling, out-of-order (o-o-o): binary compatible, exploiting ILP in hardware: BTB, ROB, Reservation Stations, ...

How far can you take it?
How much of all this complexity can you shift into the compiler?
What if you can also change instruction set architecture?

VLIW (Very Long Instruction Word)
EPIC (Explicitly Parallel Instruction Computer)
- Intel’s (and HP’s) multi-billion dollar gamble for the future of computer architecture: Itanium, IA-64
- Started ca.1994…not dead yet – but has it turned a profit?

Beyond instruction-level parallelism…
Running Example

This code adds a scalar to a vector:

\[
\text{for (i=1000; i>=0; i=i-1)} \\
\quad x[i] = x[i] + s;
\]

Assume following latency all examples

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Execution in cycles</th>
<th>Latency in cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
FP Loop: Where are the Hazards?

• First translate into MIPS code:
  - To simplify, assume 8 is lowest address

Loop:
L.D F0,0(R1) ; F0=vector element
ADD.D F4,F0,F2 ; add scalar from F2
S.D 0(R1),F4 ; store result
DSUBUI R1,R1,8 ; decrement pointer 8B (DW)
BNEZ R1,Loop ; branch R1!=zero
NOP ; delayed branch slot

Where are the stalls?
FP Loop Showing Stalls

1 Loop: L.D F0,0(R1) ;F0=vector element
2 stall
3 ADD.D F4,F0,F2 ;add scalar in F2
4 stall
5 stall
6 S.D 0(R1),F4 ;store result
7 DSUBUI R1,R1,8 ;decrement pointer 8B (DW)
8 BNEZ R1,Loop ;branch R1!=zero
9 stall ;delayed branch slot

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
</tbody>
</table>

9 clocks: Rewrite code to minimize stalls?
Revised FP Loop Minimizing Stalls

1 Loop: L.D F0,0(R1)
2 stall
3 ADD.D F4,F0,F2
4 DSUBUI R1,R1,8
5 BNEZ R1,Loop ;delayed branch
6 S.D 8(R1),F4 ;altered when move past DSUBUI

Swap BNEZ and S.D by changing address of S.D

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
</tbody>
</table>

6 clocks, but just 3 for execution, 3 for loop overhead; How make faster?
Unroll the loop four times

- Four copies of the loop body
- One copy of increment and test
- Adjust register-indirect loads using offsets

1 Loop:

1. L.D F0,0(R1)
2. ADD.D F4,F0,F2
3. S.D 0(R1),F4 ;drop DSUBUI & BNEZ
4. L.D F0,-8(R1)
5. ADD.D F4,F0,F2
6. S.D -8(R1),F4 ;drop DSUBUI & BNEZ
7. L.D F0,-16(R1)
8. ADD.D F4,F0,F2
9. S.D -16(R1),F4 ;drop DSUBUI & BNEZ
10. L.D F0,-24(R1)
11. ADD.D F4,F0,F2
12. S.D -24(R1),F4
13. DSUBUI R1,R1,#32 ;alter to 4*8
14. BNEZ R1,LOOP
15. NOP

- Re-use of registers creates WAR ("anti-dependences")
- How can we remove them?
Loop unrolling...

The original “register renaming”
Unrolled Loop That Minimizes Stalls

1. Loop: L.D F0,0(R1)
2. L.D F6,-8(R1)
3. L.D F10,-16(R1)
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

What assumptions made when moved code?

- OK to move store past DSUBUI even though changes register
- OK to move loads before stores: get right data?
- When is it safe for compiler to do such changes?
Compare: without scheduling

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

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration
Assumes R1 is multiple of 4
Static overlapping of loop bodies: “Software Pipelining”

Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations

Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (~ Tomasulo in software)
Software Pipelining Example

Before: Unrolled 3 times
1. L.D F0,0(R1)
2. ADD.D F4,F0,F2
3. S.D 0(R1),F4
4. L.D F6,-8(R1)
5. ADD.D F8,F6,F2
6. S.D -8(R1),F8
7. L.D F10,-16(R1)
8. ADD.D F12,F10,F2
9. S.D -16(R1),F12
10. DSUBUI R1,R1,#24
11. BNEZ R1,LOOP

After: Software Pipelined
1. S.D 0(R1),F4 ; Stores M[i]
2. ADD.D F4,F0,F2 ; Adds to M[i-1]
3. L.D F0,-16(R1); Loads M[i-2]
4. DSUBUI R1,R1,#8
5. BNEZ R1,LOOP

• Symbolic Loop Unrolling
  – Maximize result-use distance
  – Less code space than unrolling
  – Fill & drain pipe only once per loop
    vs. once per each unrolled iteration in loop unrolling

5 cycles per iteration
(3 if we can issue DSUBUI and BNEZ in parallel with other instrns)
Pipeline fills  
Pipeline full  
Pipeline drains
What if We Can Change the Instruction Set?

- Superscalar processors decide on the fly how many instructions to issue in each clock cycle
  - Have to check for dependences between all \( n \) pairs of instructions in a potential parallel issue packet
  - Hardware complexity of figuring out the number of instructions to issue is \( O(n^2) \)
    - Entirely doable for smallish \( n \), but tends to lead to multiple pipeline stages between fetch and issue

- Why not allow compiler to schedule instruction level parallelism explicitly?
- Format the instructions into a potential issue packet so that hardware need not check explicitly for dependences
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)

Transmeta’s Crusoe

Texas Instruments TMS320C64x

All the operations the compiler puts in the long instruction word are independent, so can be issued and can 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

VLIW example: Transmeta Crusoe

Transmeta’s Crusoe was a 5–issue VLIW processor. Instructions were dynamically translated from x86 in firmware “Code Morphing.”

Instruction encoding
Recall: Unrolled Loop that Minimizes Stalls for Scalar

1 Loop:  L.D  F0, 0(R1)  
2         L.D  F6, -8(R1)  
3         L.D  F10, -16(R1)  
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 reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op/branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>S.D 0(R1),F4</td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>S.D -16(R1),F12</td>
<td>S.D -8(R1),F8</td>
<td>ADD.D F28,F26,F2</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>S.D -32(R1),F20</td>
<td>S.D -24(R1),F16</td>
<td></td>
<td></td>
<td>DSUBUI R1,R1,#48</td>
<td>7</td>
</tr>
<tr>
<td>S.D -0(R1),F28</td>
<td>S.D -40(R1),F24</td>
<td></td>
<td></td>
<td>BNEZ R1,LOOP</td>
<td>8</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)
Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations.

Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (~ Tomasulo in SW).
Software Pipelining Example

Before: Unrolled 3 times
1. L.D F0,0(R1)
2. ADD.D F4,F0,F2
3. S.D 0(R1),F4
4. L.D F6,-8(R1)
5. ADD.D F8,F6,F2
6. S.D -8(R1),F8
7. L.D F10,-16(R1)
8. ADD.D F12,F10,F2
9. S.D -16(R1),F12
10. DSUBUI R1,R1,#24
11. BNEZ R1,LOOP

After: Software Pipelined (single-issue)
1. S.D 0(R1),F4 ; Stores M[i]
2. ADD.D F4,F0,F2 ; Adds to M[i-1]
3. L.D F0,-16(R1); Loads M[i-2]
4. DSUBUI R1,R1,#8
5. BNEZ R1,LOOP

• Symbolic Loop Unrolling
  - Maximize result-use distance
  - Less code space than unrolling
  - Fill & drain pipe only once per loop
    vs. once per each unrolled iteration in loop unrolling
Software Pipelining with Loop Unrolling in VLIW

Software pipelined across 9 iterations of original loop

- In each iteration of above loop, we:
  - Store to m,m-8,m-16 (iterations I-3,I-2,I-1)
  - Compute for m-24,m-32,m-40 (iterations I,I+1,I+2)
  - Load from m-48,m-56,m-64 (iterations I+3,I+4,I+5)

9 results in 9 cycles, or 1 clock per iteration

Average: 3.67 instrs per clock, 91.75% efficiency

Note: Need fewer registers for software pipelining
(only using 7 registers here, was using 15)
Trace Scheduling

Parallelism across IF branches vs. LOOP branches?

Two steps:

- **Trace Selection**
  - Find likely sequence of basic blocks (trace) of (statically predicted or profile predicted) long sequence of straight-line code

- **Trace Compaction**
  - Squeeze trace into few VLIW instructions
  - Need bookkeeping code in case prediction is wrong

This is a form of compiler-generated speculation

- Compiler must generate “fixup” code to handle cases in which trace is not the taken branch
- Needs extra registers: undoes bad guess by discarding

Subtle compiler bugs mean wrong answer vs. poorer performance; no hardware interlocks

Some scope for support from instruction set to help with fixup/rollback

- Eg Transmeta’s shadow registers, store gate
Speculation: HW (Tomasulo) vs. SW (VLIW)

**HW advantages:**
- HW better at memory disambiguation since actual addresses are known at runtime
- HW better at branch prediction since lower overhead
- HW maintains precise exception model
- HW does not execute bookkeeping instructions
- Same software works across multiple implementations
- Smaller code size (not as many nops filling blank instructions)

**SW advantages:**
- Window of instructions that is examined for parallelism is unlimited
- Reduced transistor count, and power consumption (?)
- More elaborate types of speculation can be easier?
- Speculation can be based on large-scale program behavior, not just local information
Problems with “First Generation” VLIW

- Increase in code size
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding

- Operated in lock-step; no hazard detection HW
  - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might know functional unit latencies, but caches harder to predict

- Binary code compatibility
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code
IA-64: Intel’s bid to create a new instruction set architecture

- EPIC = “2nd generation VLIW”? 
- ISA exposes parallelism (and many other issues) to the compiler
- But is binary-compatible across processor implementations

Itanium™ first implementation (2001)
- 6-wide, 10-stage pipeline

Itanium 2 (2002-2010)
- 6-wide, 8-stage pipeline

Itanium 9500 (Poulson) (2012)
- 12-wide, 11-stage pipeline

(2017: Kittson, but no details available)
Instruction group: a sequence of consecutive instructions with no register data dependences

- All instructions in a group could be executed in parallel, if sufficient hardware resources exist and if any dependences through memory are preserved
- Instruction group can be arbitrarily long, but compiler must explicitly indicate boundary between one instruction group and another by placing a **stop** between 2 instructions that belong to different groups

IA-64 instructions are encoded in bundles, which are 128 bits wide.

Each bundle consists of a 5-bit template field and 3 instructions, each 41 bits in length

One purpose of the template field is to mark where instructions in the bundle are dependent or independent, and whether they can be issued in parallel with the next bundle

Eg for Poulson, groups of up to 4 bundles can be issued in parallel

Smaller code size than old VLIW, larger than x86/RISC
Instruction bundling in IA-64

Instructions can be explicitly sequential:

- `add r1 = r2, r3 ;;`
- `sub r4 = r1, r2 ;;`
- `shl r2=r4,r8`

Or not:

- `add r1 = r2, r3`
- `sub r4 = r11, r21`
- `shl r12 = r14, r8 ;;`

The “;;” syntax sets the “stop” bit that marks the end of a sequence of bundles that can be issued in parallel.
To help trace scheduling and software pipelining, the Itanium instruction set includes several interesting mechanisms:

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

Job creation scheme for compiler engineers

We will look at several of these in more detail ....
General-purpose registers are configured to help accelerate procedure calls using a *register stack*.

- mechanism similar to that developed in the Berkeley RISC-I processor and used in the SPARC architecture.
- Registers 0-31 are always accessible and addressed as 0-31.
- Registers 32-128 are used as a register stack and each procedure is allocated a set of registers (from 0 to 96).
- The new register stack frame is created for a called procedure by renaming the registers in hardware;
- a special register called the current frame pointer (CFM) points to the set of registers to be used by a given procedure.

**IA-64 register stack**

```
g()  
f()  
c()  
f calls g(d,e,f)  
main calls f(a,b,c)  
```
64 1-bit predicate registers

(p1) add r1 = r2, r3
(p2) sub r1 = r2, r3 ;;
shl r12 = r1, r8

Predication means

- Compiler can move instructions across conditional branches
- To pack parallel issue groups
- May also eliminate some conditional branches completely
- Avoiding branch prediction and misprediction

There are also 8 64-bit Branch registers used to hold branch destination addresses for indirect branches – see later
IA64 load instruction variants

- IA64 has several different mechanisms to enable the compiler to schedule loads

- ld.s – speculative, non-faulting

- ld.a – speculative, “advanced” – checks for aliasing stores

- Register values may be marked “NaT” – not a thing
  - If speculation was invalid

- Advanced Load Address Table (ALAT) tracks stores to addresses of “advanced” loads

http://www.stanford.edu/class/ee382a/handouts/L13-Vector.pdf
**IA64: Speculative, Non-Faulting Load**

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

[Diagram showing speculative load and branch with compensation code](http://www.stanford.edu/class/ee382a/handouts/L13-Vector.pdf)
Speculatively-loaded data can be consumed prior to check
“speculation” status is propagated with speculated data
Any instruction that uses a speculative result also becomes speculative
(i.e. suppressed exceptions)
chk.s checks the entire dataflow sequence for exceptions
**IA64: Speculative “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.
Static branch hints can be encoded with every branch
- taken vs. not-taken
- whether to allocate (or deallocate) an entry in the dynamic BP hardware
  (Itanium-1 used a 512-entry 2-level BHT and 64-entry BTB)

TAR (Target Address Register)
- a small, fully-associative BTAC-like structure
- contents are controlled entirely by a “prepare-to-branch” (aka an “importance” hint bit)
- a hit in TAR overrides all other predictions

RSB (Return Address Stack)
- Procedure return addr is pushed (or popped) when a procedure is called (or when it returns)
- Predicts nPC when executing register-indirect branches

ISA provides for separate storages for “temporal” vs “non-temporal” data, each with its own multiple level of hierarchies

Load and Store instructions can give hints about where cached copies should be held after a cache miss
Both the integer and floating point registers support register rotation for registers 32-128.

Register rotation is designed to ease the task of register allocation in software pipelined loops.

When combined with predication, possible to avoid the need for unrolling and for separate prologue and epilogue code for a software pipelined loop.

makes the SW-pipelining usable for loops with smaller numbers of iterations, where the overheads would traditionally negate many of the advantages.
How Register Rotation Helps Software Pipelining

The concept of a software pipelining branch:

```
L1:    ld4   r35 = [r4], 4    // post-increment by 4
       st4   [r5] = r37, 4    // post-increment by 4
       br.ctop L1 ;;
```

The `br.ctop` instruction in the example rotates the general registers (actually `br.ctop` does more as we shall see)

Therefore the value stored into `r35` is read in `r37` two iterations (and two rotations) later.

The register rotation eliminated a dependence between the load and the store instructions, and allowed the loop to execute in one cycle.

- Register rotation is useful for procedure calls
- It’s also useful for software-pipelined loops
- The logical-to-physical register mapping is shifted by 1 each time the branch ("br.ctop") is executed
Software Pipelining Example in the IA-64

```
mov pr.rot = 0        // Clear all rotating predicate registers
cmp.eq p16,p0 = r0,r0 // Set p16=1
mov ar.lc = 4         // Set loop counter to n-1
mov ar.ec = 3         // Set epilog counter to 3

loop:
    ...                  
(p16) ld r32 = [r12], 1 // Stage 1: load x
(p17) add r34 = 1, r33 // Stage 2: y=x+1
(p18) stl [r13] = r35,1 // Stage 3: store y
    br.ctop loop         // Branch back
```

“Stage” predicate mechanism allows successive stages of the software pipeline to be filled on start-up and drained when the loop terminates.

The software pipeline branch “br.ctop” rotates the predicate registers, and injects a 1 into p16.

Thus enabling one stage at a time, for execution of prologue.

When loop trip count is reached, “br.ctop” injects 0 into p16, disabling one stage at a time, then finally falls-through.
Software Pipelining Example in the IA-64

loop:
(p16) ld l r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35,1
br.ctop loop

General Registers (Physical)
32 33 34 35 36 37 38 39

General Registers (Logical)

Predicate Registers
1 0 0
16 17 18

RRB

LC
4

EC
3

RRB
0

Memory
x1
x2
x3
x4
x5

http://www.cs.ualberta.ca/~amaral/courses/680/webslides/TF-HWSupSoftPipeline/
Software Pipelining Example in the IA-64

```
loop: (p16)  ldI  r32  = [r12], 1
        (p17)  add r34   = 1, r33
        (p18)  stl [r13] = r35,1
        br.ctop loop
```
Software Pipelining Example in the IA-64

loop:
(p16)  ld1 r32 = [r12], 1
(p17)  add r34 = 1, r33
(p18)  stl [r13] = r35,1
br.ctop loop

General Registers (Physical)

32 33 34 35 36 37 38 39
x1

General Registers (Logical)

32 33 34 35 36 37 38 39

Predicate Registers

1 0 0
16 17 18

LC 4
EC 3

RRB 0
Software Pipelining Example in the IA-64

Loop:
(p16) ld1 r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35,1
br.ctop loop
Software Pipelining Example in the IA-64

Loop:
(p16) ld1 r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) st1 [r13] = r35,1
br.c.top loop

General Registers (Physical)

<table>
<thead>
<tr>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
</tr>
</thead>
</table>

| 33 | 34 | 35 | 36 | 37 | 38 | 39 | 32 |

General Registers (Logical)

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

Predicate Registers

<table>
<thead>
<tr>
<th>16</th>
<th>17</th>
<th>18</th>
</tr>
</thead>
</table>

LC

<table>
<thead>
<tr>
<th>3</th>
</tr>
</thead>
</table>

EC

<table>
<thead>
<tr>
<th>3</th>
</tr>
</thead>
</table>

RRB

| -1  |
Software Pipelining Example in the IA-64

loop:
(p16) ld1 r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35, 1
br.ctop loop
Software Pipelining Example in the IA-64

loop:
(p16) ld1 r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35,1
br.ctop loop

General Registers (Physical)
32 33 34 35 36 37 38 39
x1 y1
33 34 35 36 37 38 39 32

General Registers (Logical)
x2

Predicate Registers
1 1 0
16 17 18

LC
EC
RRB
3
3
-1
Software Pipelining Example in the IA-64

**Predicate Registers**

```
<table>
<thead>
<tr>
<th>LC</th>
<th>EC</th>
<th>RRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>-1</td>
</tr>
</tbody>
</table>
```

**General Registers (Physical)**

```
<table>
<thead>
<tr>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>y1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**General Registers (Logical)**

```
<table>
<thead>
<tr>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>x2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Memory**

```
<table>
<thead>
<tr>
<th>x1</th>
<th>x2</th>
<th>x3</th>
<th>x4</th>
<th>x5</th>
</tr>
</thead>
</table>
```

**Loop:**

(p16) \( \text{ldl \ r32 = [r12], 1} \)

(p17) \( \text{add \ r34 = 1, r33} \)

(p18) \( \text{stl [r13] = r35, 1} \)

\( \text{br.ctop loop} \)
Software Pipelining Example in the IA-64

Loop:
(p16) ld1 r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35, 1
br.ctop loop
Software Pipelining Example in the IA-64

**Loop:**

(p16) $\text{ldl } r32 = [r12], 1$

(p17) $\text{add } r34 = 1, r33$

(p18) $\text{stl } [r13] = r35, 1$

**Predicate Registers**

**General Registers (Physical)**

<table>
<thead>
<tr>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>x1</td>
<td>y1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x2</td>
</tr>
</tbody>
</table>

**General Registers (Logical)**

| 34 | 35 | 36 | 37 | 38 | 39 | 32 | 33 |

**Predicate Registers**

<table>
<thead>
<tr>
<th>16</th>
<th>17</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Memory**

- **x1**
- **x2**
- **x3**
- **x4**
- **x5**
Software Pipelining Example in the IA-64

Loop:
(p16) `ldl r32 = [r12], 1`
(p17) `add r34 = 1, r33`
(p18) `stl [r13] = r35, 1`
`br.ctop loop`

Memory

General Registers (Physical)

<table>
<thead>
<tr>
<th></th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>y1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

General Registers (Logical)

<table>
<thead>
<tr>
<th></th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
<th>32</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Predicate Registers

<table>
<thead>
<tr>
<th></th>
<th>16</th>
<th>17</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

RRB

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

LC

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

EC

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

RRB

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td></td>
</tr>
</tbody>
</table>
Software Pipelining Example in the IA-64

loop:
(p16) ldль r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) std [r13] = r35,1
br.ctop loop

Memory

x1
x2
x3
x4
x5

General Registers (Physical)
32 33 34 35 36 37 38 39
y2 y1
34 35 36 37 38 39 32 33

General Registers (Logical)

Predicate Registers

1 1 1
16 17 18

LC
2
EC
3
RRB
-2

Advanced Computer Architecture Chapter 5.69
Software Pipelining Example in the IA-64

Loop:
(p16) ld r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35, 1
br.ctop loop
Software Pipelining Example in the IA-64

loop:
(p16) ld1 r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35, 1
br.ctop loop
Some hidden slides are not in handout

We continue with start of pipeline drain phase
Software Pipelining Example in the IA-64

**Loop:**
(p16) \( \text{ldl } r32 = [r12], 1 \)
(p17) \( \text{add } r34 = 1, r33 \)
(p18) \( \text{stl } [r13] = r35, 1 \)

**br.ctop loop**

**Memory**
- \( x1 \)
- \( x2 \)
- \( x3 \)
- \( x4 \)
- \( x5 \)

**General Registers (Physical)**
- 32 33 34 35 36 37 38 39
- y2 y1 x5 x4 y4 y3
- 36 37 38 39 32 33 34 35

**General Registers (Logical)**
- y1
- y2
- y3

**Predicate Registers**
- LC 0
- EC 3
- RRB -4
Software Pipelining Example in the IA-64

Loop:
(p16) `ldl r32 = [r12], 1`
(p17) `add r34 = 1, r33`
(p18) `stl [r13] = r35, 1`
`br.ctop loop`

Memory

General Registers (Physical)

<table>
<thead>
<tr>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
<th>37</th>
<th>38</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>y2</td>
<td>y1</td>
<td>x5</td>
<td>x4</td>
<td>y4</td>
<td>y3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

General Registers (Logical)

<table>
<thead>
<tr>
<th>37</th>
<th>38</th>
<th>39</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
<th>36</th>
</tr>
</thead>
</table>

Predicate Registers

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
</tbody>
</table>

LC

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
</table>

EC

|   2  |

RRB

|  -5  |
Software Pipelining Example in the IA-64

**Predicate Registers**
- LC: 0
- EC: 1
- RRB: -6

**General Registers (Logical)**
- x5, y5, y4, y3
- y2, y1

**General Registers (Physical)**
- 0, 0, 1
- 16, 17, 18

**Memory**
- x1, x2, x3, x4, x5

**Loop:**
(p16) ld1 r32 = [r12], 1
(p17) add r34 = 1, r33
(p18) stl [r13] = r35, 1
    br.ctop loop
Software Pipelining Example in the IA-64

Loop:
(p16)  ld1 r32 = [r12], 1
(p17)  add r34 = 1, r33
(p18)  stl [r13] = r35,1
br.ctop loop

Memory

General Registers (Physical)
32  33  34  35  36  37  38  39
y2  y1  x5  y5  y4  y3
37  38  39  32  33  34  35  36
General Registers (Logical)

Predicate Registers
0  0  0
16  17  18

LC
0
EC
0
RRB
-7
Caches
- 32KB L1 (2 cycle)
- 96KB L2 (7 cycle)
- 2 or 4 MB L3 (off chip)

133 MHz 64-bit bus

SpecFP: 711

SpecInt: 404

36-bit addresses (64GB)
Advanced Computer Architecture Chapter 5.

Itanium™ EPIC Design Maximizes SW-HW Synergy
(Copyright: Intel at Hotchips ’00)

Architecture Features programmed by compiler:

<table>
<thead>
<tr>
<th>Branch Hints</th>
<th>Explicit Parallelism</th>
<th>Register Stack &amp; Rotation</th>
<th>Predication</th>
<th>Data &amp; Control Speculation</th>
<th>Memory Hints</th>
</tr>
</thead>
</table>

Micro-architecture Features in hardware:

**Fetch**
- Instruction Cache & Branch Predictors

**Issue**
- Fast, Simple 6-Issue

**Register Handling**
- 128 GR & 128 FR, Register Remap & Stack Engine

**Control**
- Bypasses & Dependencies

**Parallel Resources**
- 4 Integer + 4 MMX Units
- 2 FMACs (4 for SSE)
- 2 L.D/ST units
- 32 entry ALAT

**Memory Subsystem**
- Three levels of cache: L1, L2, L3
10 Stage In-Order Core Pipeline
(Copyright: Intel at Hotchips ’00)

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
Itanium 2

- Caches
  - 32KB L1 (1 cycle)
  - 256KB L2 (5 cycle)
  - 3 MB L3 (on chip)

- 200 MHz 128-bit Bus

- SpecFP: 1356
- SpecInt: 810
- 44-bit addresses (18TB)

- 221M transistors
- 19.5 x 21.6 mm

- [http://cpus.hp.com/technical_references/](http://cpus.hp.com/technical_references/)
Comments on Itanium

Remarkably, the Itanium has many of the features more commonly associated with the dynamically-scheduled pipelines:

- strong emphasis on branch prediction
- register renaming,
- scoreboard,
- a deep pipeline with many stages before execution (to handle instruction alignment, renaming, etc.),
- several stages following execution to handle exception detection

Surprising that an approach whose goal is to rely on compiler technology and simpler HW seems to be at least as complex as dynamically scheduled processors!
Comments on Itanium

Compare Itanium II

<table>
<thead>
<tr>
<th>IPG</th>
<th>FET</th>
<th>ROT</th>
<th>EXP</th>
<th>REN</th>
<th>WL.D</th>
<th>REG</th>
<th>EXE</th>
<th>DET</th>
<th>WRB</th>
</tr>
</thead>
<tbody>
<tr>
<td>INST POINTER GENERATION</td>
<td>FETCH</td>
<td>ROTATE</td>
<td>EXECUTE</td>
<td>EXCEPTION DETECT</td>
<td>WRITE-BACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With IBM Power 4:

[Diagram of IBM Power 4 architecture]

http://ixbtlabs.com/articles/ibmpower4/
EPIC/IA-64/Itanium principles

- **Start loads early**
  - advance loads - move above stores when alias analysis is incomplete
  - speculative loads - move above branches

- **Predication to eliminate many conditional branches**
  - 64 predicate registers
  - almost every instruction is predicated

- **Register rich**
  - 128 integer registers (64 bits each)
  - 128 floating-point registers

- **Independence architecture**
  - VLIW flavor, but fully interlocked (i.e., no delay slots)
  - three 41-bit instruction syllables per 128-bit "bundle"
  - each bundle contains 5 "template bits" which specify independence of following syllables (within bundle and between bundles)

- **Unbundled branch architecture**
  - eight branch registers
  - multiway branches

- **Rotating register files**
  - lower 48 of the predicate registers rotate
  - lower 96 of the integer registers rotate
Itanium Timeline

1981: **Bob Rau** leads Polycyclic Architecture project at TRW/ESL
1983: **Josh Fisher** describes ELI-512 VLIW design and trace scheduling
1983-1988: Rau at Cydrome works on VLIW design called Cydra-5, but company folds 1988
1984-1990: **Fisher at Multiflow works on VLIW design** called Trace, but company folds 1990
1988: Dick Lampman at HP hires Bob Rau and **Mike Schlansker** from Cydrome and also gets IP rights from Cydrome
1989: Rau & Schlansker begin FAST (Fine-grained Architecture & Software Technologies) research project at HP; later develop HP PlayDoh architecture
1990-1993: Bill Worley leads PA-WW (Precision Architecture Wide-Word) effort at HP Labs to be successor to PA-RISC architecture; also called SP-PA (Super-Parallel Processor Architecture) & SWS (SuperWorkStation)

HP hires Josh Fisher, input to PA-WW
Input to PA-WW from Hitachi team, led by Yasuyuki Okada
1991: Hans Mulder joins Intel to start work on a 64-bit architecture
1992: Worley recommends HP seek a semiconductor manufacturing partner
1993: HP starts effort to develop PA-WW as a product
Dec 1993: HP investigates partnership with Intel
June 1994: announcement of cooperation between HP & Intel; PA-WW starting point for joint design; John Crawford of Intel leads joint team
1997: the term EPIC is coined
Oct 1997: Microprocessor Forum presentations by Intel and HP
Feb 1999: release of ISA details of IA-64
2001: Intel marketing prefers IPF (Itanium Processor Family) to IA-64
May 2001 - Itanium (Merced)
July 2002 - Itanium 2 (McKinley)
Aug 2004: “Itanium sales fall $13.4bn shy of $14bn forecast” (The Register)
Dec 2004: HP transfers last of Itanium development to Intel

(based on [http://www.cs.clemson.edu/~mark/epic.html](http://www.cs.clemson.edu/~mark/epic.html))
• NASA's 10,240-processor Columbia supercomputer (2004-2013) built from 20 Altix systems, each powered by 512 Intel Itanium 2 processors. Peak performance 42.7 TeraFlops; 20TB RAM. Single shared address space. Ran Linux. (Image courtesy of Silicon Graphics, Inc.)
  
  
  • http://www.nas.nasa.gov/hecc/resources/columbia.html
# Top 20 SPEC systems

<table>
<thead>
<tr>
<th>#</th>
<th>MHz</th>
<th>Processor</th>
<th>int peak</th>
<th>int base</th>
<th>Full results</th>
<th>MHz</th>
<th>Processor</th>
<th>fp peak</th>
<th>fp base</th>
<th>Full results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2933</td>
<td>Core 2 Duo EE</td>
<td>3119</td>
<td>3108</td>
<td>HTML</td>
<td>2300</td>
<td>POWER5+</td>
<td>3642</td>
<td>3369</td>
<td>HTML</td>
</tr>
<tr>
<td>2</td>
<td>3000</td>
<td>Xeon 51xx</td>
<td>3102</td>
<td>3089</td>
<td>HTML</td>
<td>1600</td>
<td>DC Itanium 2</td>
<td>3098</td>
<td>3098</td>
<td>HTML</td>
</tr>
<tr>
<td>3</td>
<td>2666</td>
<td>Core 2 Duo</td>
<td>2848</td>
<td>2844</td>
<td>HTML</td>
<td>3000</td>
<td>Xeon 51xx</td>
<td>3056</td>
<td>2811</td>
<td>HTML</td>
</tr>
<tr>
<td>4</td>
<td>2660</td>
<td>Xeon 30xx</td>
<td>2835</td>
<td>2826</td>
<td>HTML</td>
<td>2933</td>
<td>Core 2 Duo EE</td>
<td>3050</td>
<td>3048</td>
<td>HTML</td>
</tr>
<tr>
<td>5</td>
<td>3000</td>
<td>Opteron</td>
<td>2119</td>
<td>1942</td>
<td>HTML</td>
<td>2660</td>
<td>Xeon 30xx</td>
<td>3044</td>
<td>2763</td>
<td>HTML</td>
</tr>
<tr>
<td>6</td>
<td>2800</td>
<td>Athlon 64 FX</td>
<td>2061</td>
<td>1923</td>
<td>HTML</td>
<td>1600</td>
<td>Itanium 2</td>
<td>3017</td>
<td>3017</td>
<td>HTML</td>
</tr>
<tr>
<td>7</td>
<td>2800</td>
<td>Opteron AM2</td>
<td>1960</td>
<td>1749</td>
<td>HTML</td>
<td>2667</td>
<td>Core 2 Duo</td>
<td>2850</td>
<td>2847</td>
<td>HTML</td>
</tr>
<tr>
<td>8</td>
<td>2300</td>
<td>POWER5+</td>
<td>1900</td>
<td>1820</td>
<td>HTML</td>
<td>1900</td>
<td>POWER5</td>
<td>2796</td>
<td>2585</td>
<td>HTML</td>
</tr>
<tr>
<td>9</td>
<td>3733</td>
<td>Pentium 4 E</td>
<td>1872</td>
<td>1870</td>
<td>HTML</td>
<td>3000</td>
<td>Opteron</td>
<td>2497</td>
<td>2260</td>
<td>HTML</td>
</tr>
<tr>
<td>10</td>
<td>3800</td>
<td>Pentium 4 Xeon</td>
<td>1856</td>
<td>1854</td>
<td>HTML</td>
<td>2800</td>
<td>Opteron AM2</td>
<td>2462</td>
<td>2230</td>
<td>HTML</td>
</tr>
<tr>
<td>11</td>
<td>2260</td>
<td>Pentium M</td>
<td>1839</td>
<td>1812</td>
<td>HTML</td>
<td>3733</td>
<td>Pentium 4 E</td>
<td>2283</td>
<td>2280</td>
<td>HTML</td>
</tr>
<tr>
<td>12</td>
<td>3600</td>
<td>Pentium D</td>
<td>1814</td>
<td>1810</td>
<td>HTML</td>
<td>2800</td>
<td>Athlon 64 FX</td>
<td>2261</td>
<td>2086</td>
<td>HTML</td>
</tr>
<tr>
<td>13</td>
<td>2167</td>
<td>Core Duo</td>
<td>1804</td>
<td>1796</td>
<td>HTML</td>
<td>2700</td>
<td>PowerPC 970MP</td>
<td>2259</td>
<td>2060</td>
<td>HTML</td>
</tr>
<tr>
<td>14</td>
<td>3600</td>
<td>Pentium 4</td>
<td>1774</td>
<td>1772</td>
<td>HTML</td>
<td>2160</td>
<td>SPARC64 V</td>
<td>2236</td>
<td>2094</td>
<td>HTML</td>
</tr>
<tr>
<td>15</td>
<td>3466</td>
<td>Pentium 4 EE</td>
<td>1772</td>
<td>1701</td>
<td>HTML</td>
<td>3730</td>
<td>Pentium 4 Xeon</td>
<td>2150</td>
<td>2063</td>
<td>HTML</td>
</tr>
<tr>
<td>16</td>
<td>2700</td>
<td>PowerPC 970MP</td>
<td>1706</td>
<td>1623</td>
<td>HTML</td>
<td>3600</td>
<td>Pentium D</td>
<td>2077</td>
<td>2073</td>
<td>HTML</td>
</tr>
<tr>
<td>17</td>
<td>2600</td>
<td>Athlon 64</td>
<td>1706</td>
<td>1612</td>
<td>HTML</td>
<td>3600</td>
<td>Pentium 4</td>
<td>2015</td>
<td>2009</td>
<td>HTML</td>
</tr>
<tr>
<td>18</td>
<td>2000</td>
<td>Pentium 4 Xeon LV</td>
<td>1668</td>
<td>1663</td>
<td>HTML</td>
<td>2600</td>
<td>Athlon 64</td>
<td>1829</td>
<td>1700</td>
<td>HTML</td>
</tr>
<tr>
<td>19</td>
<td>2160</td>
<td>SPARC64 V</td>
<td>1620</td>
<td>1501</td>
<td>HTML</td>
<td>1700</td>
<td>POWER4+</td>
<td>1776</td>
<td>1642</td>
<td>HTML</td>
</tr>
<tr>
<td>20</td>
<td>1600</td>
<td>Itanium 2</td>
<td>1590</td>
<td>1590</td>
<td>HTML</td>
<td>3466</td>
<td>Pentium 4 EE</td>
<td>1724</td>
<td>1719</td>
<td>HTML</td>
</tr>
</tbody>
</table>

## With Auto-parallelisation

<table>
<thead>
<tr>
<th>#</th>
<th>MHz</th>
<th>Processor</th>
<th>int peak</th>
<th>int base</th>
<th>Full results</th>
<th>MHz</th>
<th>Processor</th>
<th>fp peak</th>
<th>fp base</th>
<th>Full results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2100</td>
<td>POWER5+</td>
<td>4051</td>
<td>3210</td>
<td>HTML</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3000</td>
<td>Opteron</td>
<td>3538</td>
<td>2851</td>
<td>HTML</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2600</td>
<td>Opteron AM2</td>
<td>3338</td>
<td>2711</td>
<td>HTML</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1200</td>
<td>UltraSPARC III Cu</td>
<td>1344</td>
<td>1074</td>
<td>HTML</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary#1: Hardware versus Software Speculation Mechanisms

To speculate extensively, must be able to disambiguate memory references
- Much easier in HW than in SW for code with pointers

HW-based speculation works better when control flow is unpredictable, and when HW-based branch prediction is superior to SW-based branch prediction done at compile time
- Mispredictions mean wasted speculation

HW-based speculation maintains precise exception model even for speculated instructions

HW-based speculation does not require compensation or bookkeeping code
Summary#2: Hardware versus Software Speculation Mechanisms cont’d

- Compiler-based approaches may benefit from the ability to see further in the code sequence, resulting in better code scheduling.

- HW-based speculation with dynamic scheduling does not require different code sequences to achieve good performance for different implementations of an architecture. May be the most important in the long run?

Example: ARM’s “big.LITTLE” architecture

- Multicore processor with a mixture of large out-of-order cores (A15) and small in-order cores (A7) (eg Exynos 5 Octa in Samsung Galaxy S4).

- Compiler is configured to schedule for in-order, assuming the out-of-order processor is less sensitive to instruction scheduling.
Summary #3: Software Scheduling

Instruction Level Parallelism (ILP) found either by compiler or hardware.

Loop level parallelism is easiest to see
- SW dependencies/compiler sophistication determine if compiler can unroll loops
- Memory dependencies hardest to determine => Memory disambiguation
- Very sophisticated transformations available

Trace scheduling to parallelize if statements

Superscalar and VLIW: CPI < 1 (IPC > 1)
- Dynamic issue vs. Static issue
- More instructions issue at same time => larger hazard penalty
- Limitation is often number of instructions that you can successfully fetch and decode per cycle
Beyond ILP: Multithreading, Simultaneous Multithreading (SMT)

Cray/Tera MTA

- http://www.utc.edu/~jdumas/cs460/papersfa01/craymta/

- Problem: Dependencies limit sustainable throughput of single instruction stream
- Solution: Interleave execution of two or more instruction streams on same hardware to increase utilization

Instruction Issue

Reduced function unit utilization due to dependencies

Superscalar Issue

Superscalar leads to more performance, but lower utilization

Predicated Issue

Adds to function unit utilization, but results are thrown away

Chip Multiprocessor

Limited utilization when only running one thread

Fine Grained Multithreading

Intra-thread dependencies still limit performance

Simultaneous Multithreading

Maximum utilization of function units by independent operations
Different threads in different cycles: “FGMT”

Dynamic scheduling of operations from a pool of threads: “SMT”

Reduced function unit utilization due to dependencies

Limited utilization when only running one thread

Intra-thread dependencies still limit performance

Maximum utilization of function units by independent operations
Alpha 21464
One CPU with 4 Thread Processing Units (TPUs)
“6% area overhead over single-thread 4-issue CPU”
Multiprogrammed workload

SMT performance

Alpha 21464

Intel Pentium 4 with hyperthreading:

SMT in the Intel Atom

Intel’s bid to steal back some of the low-power market for IA-32 and Windows

- In-order
- 2-way SMT
- 2 instructions per cycle (from same or different threads)
Each thread runs slow?

- The point of Simultaneous Multithreading is that resources are dynamically assigned, so if only one thread can run it can run faster

SMT threads contend for resources

- Possibly symbiotically?
  - One thread is memory-intensive, one arithmetic-intensive?
- Possibly destructively
  - Thrashing the cache? Other shared resources... (TLB?)

Which resources should be partitioned per-thread, and which should be shared on-demand?

SMT threads need to be scheduled fairly

- Can one thread monopolise the whole CPU?
  - Denial of service risk
  - Slow thread that suffers lots of cache misses fills RUU and blocks issue
SMT – latency-hiding

SMT threads exploit memory-system parallelism
- Easy way to get lots of memory accesses in-flight
- “Latency hiding” – overlapping data access with compute

What limits the number of threads we can have?

SMT threads need a *lot* of registers
- A lot of logical registers – but they share physical registers?

In a machine *without* register renaming
- What about *statically* partitioning the register file based on the number of registers each thread actually needs?
- This is what many GPUs do
- Leads to tradeoff: lots of lightweight threads to maximise latency hiding? Or fewer heavyweight threads that benefit from lots of registers?
- Nvidia and AMD call this “occupancy”
Mapping threads into the register file

- If each thread needs few registers, we can have lots of them co-existing in the same physical register file.
- Alternatively, we could have fewer, fatter threads.
- More threads = higher “occupancy”
- Better latency hiding
- Tricky tradeoff!
CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1) Select Compute Capability (click): 3.5

1.b) Select Shared Memory Size Config (bytes) 49152

2) Enter your resource usage:

<table>
<thead>
<tr>
<th>Threads Per Block</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers Per Thread</td>
<td>32</td>
</tr>
<tr>
<td>Shared Memory Per Block (bytes)</td>
<td>4</td>
</tr>
</tbody>
</table>

(Don’t edit anything below this line)

3) GPU Occupancy Data is displayed here and in the graphs:

| Active Threads per Multiprocessor | 1024 |
| Active Warps per Multiprocessor | 32 |
| Active Thread Blocks per Multiprocessor | 16 |
| Occupancy of each Multiprocessor | 50% |

Physical Limits for GPU Compute Capability:

| Threads per Warp | 32 |
| Warps per Multiprocessor | 64 |
| Threads per Multiprocessor | 2048 |
| Total # of 32-bit registers per Multiprocessor | 65536 |
| Register allocation unit size | 256 |
| Warp allocation granularity | warp |
| Registers per Thread | 256 |
| Shared Memory per Multiprocessor (bytes) | 49152 |
| Shared Memory Allocation unit size | 256 |
| Warp allocation granularity | 4 |
| Maximum Thread Block Size | 1024 |

Allocated Resources Per Block / Limit Per SM = Allocatable Blocks Per SM

| Warps (Threads Per Block / Threads Per Warp) | 2 | 64 | 16 |
| Registers (Warp limit per SM due to per-warp reg count) | 2 | 64 | 32 |
| Shared Memory (Bytes) | 0 | 49152 | 16 |

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor

| Limited by Max Warps or Max Blocks per Multiprocessor | 16 | 2 | 32 |
| Limited by Registers per Multiprocessor | 32 |
| Limited by Shared Memory per Multiprocessor | 16 | 2 | 32 |

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64 Occupancy = 32 / 64 = 50%

Nvidia’s “CUDA Occupancy Calculator”

Shows relationship between #registers/thread and the amount of SMT available

Higher occupancy enables better hiding of memory access latency

CUDA Occupancy Calculator

Version: 5.1

Copyright and License
1.) Select Compute Capability (click): 3.5
1.b) Select Shared Memory Size Config (bytes): 49152

2.) Enter your resource usage:
   Threads Per Block: 64
   Registers Per Thread: 32
   Shared Memory Per Block (bytes): 0
   (Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:
   Active Threads per Multiprocessor: 1024
   Active Warps per Multiprocessor: 32
   Active Thread Blocks per Multiprocessor: 16
   Occupancy of each Multiprocessor: 50%

Physical Limits for GPU Compute Capability: 3.5

- Threads per Warp: 32
- Warps per Multiprocessor: 64
- Threads per Multiprocessor: 2048
- Thread Blocks per Multiprocessor: 16
- Total # of 32-bit registers per Multiprocessor: 65536
- Register allocation unit size: 256
- Register allocation granularity: warp
- Registers per Thread: 255
- Shared Memory per Multiprocessor (bytes): 49152
- Shared Memory Allocation unit size: 256
- Warp allocation granularity: 4
- Maximum Thread Block Size: 1024

Allocated Resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Per Block</th>
<th>Limit Per SM</th>
<th>Blocks Per SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warps</td>
<td>2</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>Registers</td>
<td>2</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>Shared Memory (Bytes)</td>
<td>0</td>
<td>49152</td>
<td>16</td>
</tr>
</tbody>
</table>

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor

- Limited by Max Warps or Max Blocks per Multiprocessor: 16
- Limited by Registers per Multiprocessor: 32
- Limited by Shared Memory per Multiprocessor: 16

Nvidia’s “CUDA Occupancy Calculator”

Shows relationship between #registers/thread and the amount of SMT available.

Higher occupancy enables better hiding of memory access latency.
CUDA GPU Occupancy Calculator

Impact of Varying Register Count Per Thread

- Multiprocessor Warp Occupancy (%)
- Registers Per Thread
- My Register Count 32

CUDA Occupancy Calculator shows relationship between registers/thread and the amount of available warp occupancy enabling better hiding of memory access latency.
CUDA Occupancy Calculator

1.) Select Compute Capability (click): 3.5 (Help)
1.b) Select Shared Memory Size Config... 

Multiprocessor Warp Occupancy
(# warps)

My Register Count 32x64=2048

16x128=2048

Vidia's CUD Occupancy Calculator,
Chapter summary

We have explored:

- Pipeline parallelism
- Dynamic instruction scheduling
- Static instruction scheduling
- Multiple instructions per cycle
- Very long instruction words (VLIW)
- SIMT – single-instruction multiple thread – and how it maps onto SIMD
- Multi-threading
  - Coarse-grain
  - Fine-grain
  - Simultaneous multithreading (SMT)
  - Statically-partitioned multithreading

Vector instructions and SIMD – coming soon
Is the “minimum” operator associative?

\[ \min(\min(X, Y), Z) = \min(X, \min(Y, Z)) \? \]

\[ \min(X, Y) = \text{if } X < Y \text{ then } X \text{ else } Y \]

\[ \min(\min(10, x), 100) = 100 \]
Is the “minimum” operator associative?

\[
\text{min}(\text{min}(X, Y), Z) = \text{min}(X, \text{min}(Y, Z))
\]

\[
\text{min}(X, Y) = \text{if } X < Y \text{ then } X \text{ else } Y
\]

All comparisons on NaNs always fail….

\[
\text{min}(\text{min}(10, \text{NaN}), 100) = 100
\]
Extra slides for interest/fun

Is the “minimum” operator associative?

\[ \min(\min(X, Y), Z) = \min(X, \min(Y, Z)) \] ?

\[ \min(X, Y) = \text{if } X < Y \text{ then } X \text{ else } Y \]

All comparisons on NaNs always fail….

\[ \min(X, \text{NaN}) = \text{NaN} \]
\[ \min(\text{NaN}, Y) = Y \]
\[ \min(\min(X, \text{NaN}), Y) = \min(\text{NaN}, Y) = Y \]
\[ \min(X, \min(\text{NaN}, Y)) = \min(X, Y) \]

\[ \min(\min(10, \text{NaN}), 100) = 100 \]
Associativity in floating point

(a+b)+c = a+(b+c) ?

Example: Consider 3-digit base-10 floating-point

1+1+1+1+1+1+1+......+1+1+1+1+1+1+1+1+1+1+1+1000

1000 ones

1000+1+1+1+1+1+1+1+1+......+1+1+1+1+1+1+1+1+1+1+1

1000 ones

Consequence: many compilers use loop unrolling and reassociation to enhance parallelism in summations.

And results are different!

But you can tell the compiler not to, eg:

“–fp-model precise” with Intel’s compilers

(What’s the right way to sum an array? See