A Detailed Tutorial

Version 2

Sverre Jarp
CERN - IT Division
http://nicewww.cern.ch/~sverre

11 June 1999
Change History

11 June:

Version 2

- Some editorial changes; Added date & page numbers
- Added slides on:
  - Templates; XMA-instruction;
  - Example using PMPYSHR
  - Example on Motion Estimation (MPEG2)
Global Contents

- Four distinct parts:
  - Introduction and Overview
  - Multimedia Programming
  - Floating-Point Programming
  - Optimisation
Aims

**Offer programmers**
- Comprehension of the architecture
  - Instruction set and Other features
- Capability of understanding IA-64 code
  - Compiler-generated code
  - Hand-written assembler code

**Inspiration for writing code**
- Well-targeted assembler routines
  - Highly optimised routines
- In-line assembly code
  - Full control of architectural features

Phase 1

Phase 2
Part 1

Introduction and Overview
Architectural Highlights

(Some of the) Main Innovations:

- Rich Instruction Set
- Bundled Execution
- Predicated Instructions
- Large Register Files
  - Register Stack
  - Rotating Registers
- Modulo Scheduled Loops
- Control/Data Speculation
- Cache Control Instructions
- High-precision Floating-Point
Compared to IA-32

- Many advantages:
  - Clear, explicit programming
    - After all, this is EPI C:
      - “Explicit Parallel Instruction Computing”
  - Register-based programming
    - Keep everything in registers (As long as possible)
  - Obvious register assignments
    - Integer Registers for Multimedia (Parallel Integer)
    - FP Registers for all FP work (also SIMD)
      - Exception: Integer Multiply/Divide
  - All instructions (almost) can be predicated
    - Much more general than CONDITIONAL MOVES
  - Architectural support for software pipelining
    - Modulo scheduling
Start with simple example

- Routine to initialise a floating-point value:

```plaintext
long Indx = 5  ;       // Choice may be 0 - 7
double My_fp = getval(Indx);
```

```
.proc
getval:
    alloc  r3=ar.pfs, 1, 0, 0, 0
(p0)   movl  r2=Table
(p0)   and  r32=7,r32        // Choice is 0 - 7
;;
(p0)   shladd r2=r32,4,r2    // Index table
;;
(p0)   ldfd    f8=[r2]        // Load value
(p0)   mov    ar.fps=r3
(p0)   br.ret.sptk.few b0    // return
.endp
.data
.data
Table:
real8    5.99
real8    ....
......
```

Not strictly needed for leaf routines
Initial explanation

- Lots of details
  - Many questions

```assembly
.proc
getval:
  alloc  r3=ar.pfs,R_input,R_local,R_output,R_input+R_local
  movl  r2=Table
  and   r32=7,r32  // Choice is 0 - 7
  shladd r2=r32,4,r2  // Index table
  ldfd  f8=[r2]  // Load value
  mov   ar.fps=r3
  br.ret.sptk.few b0  // return
```

Register allocation
Application registers
Enforced Bundle Break
Predicated execution
Branch return

11 June 1999
## User Register Overview

<table>
<thead>
<tr>
<th>Category</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Registers</td>
<td>128</td>
</tr>
<tr>
<td>Floating Point Registers</td>
<td>128</td>
</tr>
<tr>
<td>Predicate Registers</td>
<td>64</td>
</tr>
<tr>
<td>Branch Registers</td>
<td>8</td>
</tr>
<tr>
<td>Application Registers</td>
<td>128</td>
</tr>
<tr>
<td>CPUID D Registers</td>
<td></td>
</tr>
</tbody>
</table>

- Instruction Pointer
- User Mask
- Current Frame Marker
- NN Perf. Mon.
- Data Reg’s
CPUID D registers

- General information about the processor
  - At least 5 registers:

| CPUID D[0] | Vendor |
| CPUID D[1] | Name   |
| CPUID D[2] | Processor Serial Number |
| CPUID D[4] | Feature/Capability bits |
IA64 Common Registers

- **Integer registers**
  - 128 in total; Width is 64-bits + 1 bit (NaT); r0 = 0
  - Integer, Logical and Multimedia data

- **Floating point registers**
  - 128 in total; 82-bits wide
  - 17-bit exponent, 64-bit significand
  - f0 = 0.0; f1 = 1.0
  - Significand also used for two SIMD floats

- **Predicate registers**
  - 64 in total; 1-bit each (fire/ do not fire)
  - p0 = 1 (default value)

- **Branch registers**
  - 8 in total; 64-bits wide (for address)
Rotating Registers

- **Upper 75% rotate (when activated):**
  - General registers (r32-r127)
  - Floating Point Registers (f32-f127)
  - Predicate Registers (p16-p63)

- **Formula:**
  - Virtual Register = Physical Register - Register Rotation Base (RRB)

```
  f28 f29 f30 f31 f32 f33 f34 f35
  . . . .
  f124 f125 f126 f127
  . . . .
```

**Formula:***

Virtual Register = Physical Register - Register Rotation Base (RRB)

```
  f28 f29 f30 f31 f32 f33 f34 f35
  . . . .
  f124 f125 f126 f127
  . . . .
```
Register Stack

- The rotating integer registers serve as a stack
  - Each routine allocates via "Alloc" instruction:
    - Input + Local + Output
    - "Input + Local" may rotate (in sets of 8 registers)

Proc A

Proc B

Proc C

Proc B

Proc A

Further Calls
Which registers to use

- **Start with alloc:**
  - Alloc r36=ar.pfs,4,4,2,8
    - Available instantaneously
    - Rotate
  - Input
  - Local
  - Output

- Rotation should only be implemented
  - When input registers have been read
Instruction Types

- M
  - Memory/ Move Operations

- I
  - Complex Integer/ Multimedia Operations

- A
  - Simple Integer/ Logic/ Multimedia Operations

- F
  - Floating Point Operations (Normal/ SIMD)

- B
  - Branch Operations
### Instruction Bundle

- **'Packaging entity':**
  - 3 * 41 bit Slots for Instructions
  - 5 bits for Template
    - Typical examples: MFI or MIB
  - **A bundle of 16B:**
    - Basic unit for expressing parallelism
    - The unit that the Instruction Pointer points to
    - The unit you branch to

<table>
<thead>
<tr>
<th>Slot 0</th>
<th>Slot 1</th>
<th>Slot 2</th>
<th>T</th>
</tr>
</thead>
</table>

11 June 1999
Templates

- Decide mapping of instruction slots to execution units:
  - 12x2 basic combinations defined (out of 32)
    - Even numbers: No terminating stop-bit
    - Odd numbers: Terminating stop bit:
  - How to remember them:
    - All (except one) start w/ M:
      - Ending in I: MI, MI+I, MMI, MM+I, MFI
      - Ending in B: MI B, MMB, MFB, MBB
      - No I or B: MMF
      - Special for 64-bit immediates: MLX
    - Multiple (multiway) branches:
      - BBB

Note 1: Maximum one F instruction in a bundle

Note 2: Two templates have an embedded stop bit
No ‘unique’ format; typical examples:

- (p20) `ld4 r15=[r30],r8`
  - Load int (4 bytes) using address plus post-increment stride
- (p4) `fma.d.s0 f35=f32,f33,f127`
  - `U = X * Y + Z`
- (p2) `add r15=r3,r49,1`
  - `C = A + B + 1`

<table>
<thead>
<tr>
<th>FMA:</th>
<th>Opcode++</th>
<th>R4</th>
<th>R3</th>
<th>R2</th>
<th>R1</th>
<th>qp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Add:</th>
<th>Opcode</th>
<th>Flags</th>
<th>R3</th>
<th>R2</th>
<th>R1</th>
<th>qp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>
Instruction Types

Many Instruction Classes:

- Logical operations (e.g. and)
- Arithmetic operations (e.g. add)
- Compare operations
- Shift operations
- Multimedia operations (e.g. padd)
- Branches
- Loop controlling branches
- Floating Point operations (e.g. fma)
- SI MD Floating Point operations (e.g. fpma)
- Memory operations
- Move operations
- Cache Management operations
Conventions

- **Instruction syntax**
  - \((qp) \text{ ops[.comp_1]} r_1 = r_2, r_3\)
    - Execution is always right-to-left
    - Result(s) on left-hand side of equal-sign.
    - Almost all have a qualifying predicate
    - Many have further completers:
      - Unsigned, left, double, etc.

- **Numbering**
  - Also right-to-left

- **Immediates**
  - Various sizes exist
  - \(\text{Imm}_8\) (Signed immediate - 7 bits plus sign)

At execution time, sign bit is extended all the way to bit 63

11 June 1999
Logical Operations

- Instruction format:
  - (qp) ops \( r_1 = r_2, r_3 \)
  - (qp) ops \( r_1 = \text{Imm}_8, r_3 \)

- Valid Operations:
  - And
  - Or
  - Xor (Exclusive Or)
  - Andcm (And Complement)
    - Result\(_1 = \text{Input}_2 \& \neg\text{Input}_3 \)
Arithmetic Operations

Instruction format:

- (qp) ops$_1$ \( r_1 = r_2, r_3[,1] \)
- (qp) ops$_2$ \( r_1 = \text{Imm}_x, r_3 \)
- (qp) ops$_3$ \( r_1 = r_2, \text{count}_2, r_3 \)

Valid Operations:

- Add
- Sub
- Adds/ Addl (Imm$_{14}$, Imm$_{22}$)
- Shladd

NB: Integer multiply is a FLP operation

X86 Inc/ Dec replaced with
(qp) ops \( r_1 = r_2, r_0, 1 \)

\( Z = Y - \text{imm} \) becomes
(qp) Add \( r_1 = -\text{imm}, r_3 \)

Loading an immediate value
(qp) Add \( r_1 = \text{imm}, r_0 \)
Compare Operations

- Instruction format:
  - (qp) cmp.crel.ctype $p_1, p_2 = r_2, r_3$
  - (qp) cmp.crel.ctype $p_1, p_2 = \text{Imm}_8, r_3$
  - (qp) cmp.crel.ctype $p_1, p_2 = r_0, r_3$

- Valid Relationships:
  - Eq, ne, lt, le, gt, ge, ltu, leu, gtu, geu,

- Types:
  - None, Unc, And, Or, Or.andcm, Orcm, Andcm, And.orcm

Parallel compare instructions are discussed in the Optimisation Chapter.
Shift Operations

Instruction format:

- (qp) $\text{ops}_1$  \hspace{1cm} $r_1 = r_3, r_2$
- (qp) $\text{ops}_1[.u]$  \hspace{1cm} $r_1 = r_3, \text{count}_6$
- (qp) $\text{extr}[.u]$  \hspace{1cm} $r_1 = r_3, \text{pos}_6, \text{len}_6$
- (qp) $\text{dep}[.z]$  \hspace{1cm} $r_1 = r_2, r_3, \text{pos}_6, \text{len}_4$
- (qp) $\text{shrp}[.u]$  \hspace{1cm} $r_1 = r_3, r_2, \text{count}_6$

Valid Operations:
- $\text{ops}_1$ can be: Shl, shr, shr.u

Extract:
- Shift right and mask

Shift Right Pair can also be used for a 64-bit Rotate (Right)
Simple Multimedia

- Parallel add/ subtract
  - (qp) paddn[.sat] \( r_1 = r_2, r_3 \)
    - \( n = [1,2, \text{or } 4] \)
    - Various kinds of saturation
  - See Part 2 for further details
Floating-Point Operations

- **Standard instruction:**
  - (qp) ops.pc.sf \( f_1 = f_3, f_4, f_2 \)

- **Valid Operations:**
  - Fma \([U = X \times Y + Z]\)
  - Fms \([U = X \times Y - Z]\)
  - Fnma \([U = -(X \times Y) + Z]\)

- See part 3 for further details
SI MD Floating-Point

- Standard instruction:
  - (qp) ops.pc.sf \( f_1 = f_3, f_4, f_2 \)

- Valid Operations:
  - Fpma \( [U = X \times Y + Z] \)
  - Fpms \( [U = X \times Y - Z] \)
  - Fpnma \( [U = -(X \times Y) + Z] \)

- See part 3 for further details

NB: \( f_1 \) does NOT contain two 32-bit versions of 1.0
Load Operations

- **Standard instructions:**
  - (qp) ld.sz.ldtype.ldhint \( r_1 = [r_3], r_2 \)
  - (qp) ld.sz.ldtype.ldhint \( r_1 = [r_3], \text{Imm}_{9} \)
  - (qp) ldf.fsz.fldtype.ldhint \( f_1 = [r_3], r_2 \)
  - (qp) ldf.fsz.fldtype.ldhint \( f_1 = [r_3], \text{Imm}_{9} \)

- **Valid Sizes:**
  - **Sz:** 1/2/4/8 [bytes]
  - **Fsz:** s(ingle)/d(double)/e(extended)/8(integer)

- **Types:**
  - S/a/ sa/ c.nc/ c.clr/ c.clr.acq/ acq/ bias

Always post-modify

In the case of integer multiply (for instance)
Line Prefetch

- Place a cache-line at a given level
  - (qp) lfetch.lftype.lfhint \[ \{ r_3 \}, r_2 \]
  - (qp) lfetch.lftype.lfhint \[ \{ r_3 \}, lmm_9 \]

- Types are:
  - None
  - Fault

- Hints are:
  - None, nt1, nt2, nta
    - Note than ‘None’ means temporal level 1
    - Others: Non-temporal L1, L2, All levels

NB: There is no target
Store Operations

Standard instructions:

- (qp) st.sz.stype.sthint \[r_3\] = r_1
- (qp) st.sz.stype.sthint \[r_3\] = r_1, lmm_9
- (qp) stf.fsz.fstype.sthint \[r_3\] = f_1
- (qp) stf.fsz.fstype.sthint \[r_3\] = f_1, lmm_9

Valid Sizes:
- Same as Load

NB: Memory address is the target
No register-based post-modify
Move Operations

Between FLP and Integer:

- (qp) setf.qual \( f_1 = r_2 \)
- (qp) getf.qual \( r_1 = f_2 \)

Valid Qualifiers:

- s(ingle)/ d(double)/ exp(onent)/ sig(nificand)

NB:

- If one part of a fp register is set, the others are imposed
  - Setf.sig \( f_1 = r_2 \) sets Exponent = 0x1003E and Sign = 0.
  - [ldf8 does exactly the same]
Several different types:

- Conditional or Call branches
  - Relative offset (IP-relative) or Indirect (via branch registers)
  - Based on predication
- Return branches
  - Indirect + Qualifying Predicate (QP)
- Simple Counted Loops
  - IP-relative with AR.LC
- Modulo scheduled Counted Loop
  - IP-relative with AR.LC and AR.EC
- Modulo scheduled While Loops
  - IP-relative with QP and AR.EC
Branch syntax

- **Rather complex:**
  - (qp) Br.btype.bwh.ph.dh \( \text{target}_{25} / b_2 \)
  - (qp) Br.Call. bwh.ph.dh \( b_1 = \text{target}_{25} / b_2 \)

- **Branch Whether Hint**
  - Sptk/ spnt - Static Taken/ Not Taken
  - Dptk/ dpnt - Dynamic

- **Sequential Prefetch Hint**
  - Few/ none - few lines
  - Many

- **Branch Cache Deallocation Hint**
  - None
  - Clr
Simple Counted Loop

- **Works as ‘expected’**
  - Ar.lc counts down the loop (automatically)
  - No need to use a general register

```
Mov     ar.lc=5
Loop:   Work
        .......
        Much more work
Br.cloop.many.sptk loop
```

- **Modulo loop are more advanced**
  - Uses Epilogue Count (as well as Loop Count)
  - ... and Rotating Registers

We will deal with Modulo loops in the ‘optimisation’ chapter
Instruction Types

✓ Many Groups:
  ✓ Logical operations (e.g. and)
  ✓ Arithmetic operations (e.g. add)
  ✓ Compare operations
  ✓ Shift operations
  ✓ Multimedia operations
  ✓ Branches
  ✓ Loop controlling branches
  ✓ Floating Point operations (e.g. fma)
  ✓ SI MD Floating Point operations (e.g. fpma)
  ✓ Memory operations
  ✓ Move operations
  ✓ Cache Management operations
How to code instruction operands

Two rules:

- Assignment always on the left
  - \((qp)\) ops.qual \(r_1 = r_2, r_3\)

Mnemonics:

- Shladd \(r_1 = r_2, \text{count}_2, r_3\)
  - \text{Shift} \(r_2\) \text{Left} by \(\text{count}_2\) and \text{ADD} to \(r_3\)
- Fnma.s1 \(f_1 = f_3, f_4, f_2\)
  - \text{Flp Negative Multiply} and \text{Add}: \(f_1 = - (f_3 \times f_4) + f_2\)
- Less Obvious is: Andcm
  - \text{AND Complement}: \(r_1 = \text{Input}_2 \& \sim\text{Input}_3\)
  - Complement \text{Input}_2 or \text{Input}_3??

11 June 1999
Multimedia Overview
## User Register Overview

<table>
<thead>
<tr>
<th>Category</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Registers</td>
<td>128</td>
</tr>
<tr>
<td>Floating Point Registers</td>
<td>128</td>
</tr>
<tr>
<td>Predicate Registers</td>
<td>64</td>
</tr>
<tr>
<td>Branch Registers</td>
<td>8</td>
</tr>
<tr>
<td>Application Registers</td>
<td>128</td>
</tr>
<tr>
<td>NN CPUID Registers</td>
<td></td>
</tr>
<tr>
<td>User Mask</td>
<td></td>
</tr>
<tr>
<td>Current Frame Marker</td>
<td></td>
</tr>
<tr>
<td>Instruction Pointer</td>
<td></td>
</tr>
<tr>
<td>NN Perf. Mon. Data Reg’s</td>
<td></td>
</tr>
</tbody>
</table>

**11 June 1999**
IA64 Registers

- **Integer registers**
  - 128 in total; Width is 64-bits + 1 bit (NaT); r0 = 0
  - Integer, Logical and Multimedia data

- **Floating point registers**
  - 128 in total; 82-bits wide
  - 17-bit exponent, 64-bit mantissa
  - f0 = 0.0; f1 = 1.0
  - Mantissa also used for two SIMD floats

- **Predicate registers**
  - 64 in total; 1-bit each (fire/ do not fire)
  - p0 = 1 (default value)

- **Branch registers**
  - 8 in total; 64-bits wide (for address)
Data representation

Multimedia types have

- Three different sizes:
  - Byte: 8 * 1B (8 bits)
  - Short: 4 * 2B (16 bits)
  - Word: 2 * 4B (32 bits)

NB:

- Not all instructions handle all types!
  - Parallel add: Padd1, Padd2, Padd4
  - Parallel Sum of Absolute Differences: Psad1
## Arithmetic instructions

### Overview Table:

**Operand size**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1B</th>
<th>2B</th>
<th>4B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Padd/ Psub</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Padd.sus</td>
<td>1</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>Psub.sus</td>
<td>1</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>Pavg[.raz]</td>
<td>1</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>Pavgsub</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pshladd</td>
<td>-</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>Pshradd</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pcmp</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Pmpy</td>
<td>-</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>Pmpyshr</td>
<td>-</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>Psad</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Pmin/ Pmax</td>
<td>1</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>
### Other instructions

**Overview Table:**
- **Operand size**

<table>
<thead>
<tr>
<th>Pshl/ Pshr</th>
<th>1B</th>
<th>2B</th>
<th>4B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pshr.u</td>
<td>-</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pack.sss</th>
<th>1B</th>
<th>2B</th>
<th>4B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pack.uss</td>
<td>1B</td>
<td>2B</td>
<td>4B</td>
</tr>
<tr>
<td>Unpack</td>
<td>1B</td>
<td>2B</td>
<td>4B</td>
</tr>
</tbody>
</table>
Parallel Multiply

- \( (qp) \text{ pmpy2}.r \; r_1 = r_2, r_3 \)
  - Same instruction for left

Parallel Multiply and Shift Right

- \( (qp) \text{ pmpyshr2}[.u] \; r_1 = r_2, r_3, \text{count}_2 \)
  - Count can be: 0, 7, 15, 16

I2 and I1, respectively
Complex Multimedia - 2

- **Parallel Maximum**
  - \((qp)\) \(p_{max2}\) \(r_1 = r_2, r_3\)
  - Signed quantities
  - Unsigned if single bytes
    - \(P_{max1.u}\)

- **Parallel Sum of Absolute Differences**
  - \((qp)\) \(ps_{ad1}\) \(r_1 = r_2, r_3\)
    - Absolute difference of each sets of bytes
    - Then sum of these 8 values
Complex Multimedia - 3

- **Unpack high/low**
  - (qp) unpackn.[h l] $r_1 = r_2, r_3$
    - “High” uses bits 63-32
    - “Low” uses 31-0
    - Sizes: 1/2/4

- **Mix**
  - (qp) mixn.[l r] $r_1 = r_2, r_3$
    - “Left” uses odd-numbered pieces
    - “Right” uses even-numbered

Example 1: Unpack1.h

Example 2: Mix1.l

Both are I2

11 June 1999
Pack w/ saturation

- (qp) pack2.sat \( r_1 = r_2, r_3 \)
  - "sat" may be sss/uss
- (qp) pack4.sss \( r_1 = r_2, r_3 \)

Example of pack2
Mux2

- \((qp)\) \(mux2 \ r_1 = r_2, mbtype\)
- Very versatile
  - You ‘program’ it yourself
  - Reverse is:
    - 0x1b - 00011011 (binary)
  - Broadcast (short no. 2)
    - 0xaa - 10101010 (binary)

Mux1

- Only ‘fixed’ combinations:
  - Reverse (Bytes: 01234567)
  - Mix (73516240)
  - Shuffle (73625140)
  - Alternate (75316420)
  - Broadcast (byte 0)

I4 and I3, respectively
Parallel add/ subtract

- (qp) paddn[.sat]  \( r_1 = r_2, r_3 \)
  - Saturation of \( r_1, r_2, r_3 \) may be:
  - sss/ uus/ uuu
  - “signed” covers \( 0x80 \leftrightarrow 0x7F \)  \([0x8000 \leftrightarrow 0x7FFF]\)
  - “unsigned” covers \( 0x00 \leftrightarrow 0xFF \)  \([0x0000 \leftrightarrow 0xFFFF]\)

- Modulo arithmetic
Parallel compare

- \((qp)\) pcmpn.prel \(r_1 = r_2, r_3\)
  - One/Two/Four byte operands:
  - “Prel” may be: eq; gt (signed)
  - If true, a mask of 0xff (0xffff or 0xffffffff) is produced
  - If false, a mask of zeroes is produced
Multimedia programming

Relevant example:

- Perform 32 x 32 unsigned multiplication
  - needs: Mux, Pmpyshr, and Mix
  - 11 instructions in total
  - 7 groups

```
mux2 r34=r32,0x50
mux2 r35=r33,0x14 ;;
pmpyshr2.u r36=r34,r35,0
pmpyshr2.u r37=r34,r35,16 ;;
mix2.r r38=r37,r36
mix2.l r39=r37,r36 ;;
shr.u r40=r39,32
zxt2 r41=r39 ;;
add r42=r40,r41 ;;
shl r43=r42,16 ;;
add r31=r43,r38
```
Multimedia programming

MPEG2 motion estimation:

- From IA32 to IA64:

```assembly
Psad_top:  // 16x16 block matching
// Do PSAD for a row, accumulate results
movq mm1,[esi]
movq mm2,[esi+8]
psadbw mm1,[edi]
padbw mm2,[edi+8]
add esi, eax  // increment pointer
add edi, eax
addw mm0, mm1  // accumulate
addw mm7, mm2
dec ecx
jpe Psad_top

// 10 instructions
```

```assembly
Psad_top:  // 16x16 block matching
// Do PSAD for a row, accumulate results
ld8 r32=[r22],r21
ld8 r33=[r23],r21
ld8 r34=[r24],r21
ld8 r35=[r25],r21 ;;
psad r32=r32,r34
psad r33=r33,r35 ;;
add/padd4 r36=r36,r32
add/padd4 r37=r37,r33
Br.cloop.many.sptk Psad_top ;;

// 9 instructions, 3 groups
```
Part 3

Floating-Point Overview
# User Register Overview

<table>
<thead>
<tr>
<th>Category</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Registers</td>
<td>128</td>
</tr>
<tr>
<td>Floating Point Registers</td>
<td>128</td>
</tr>
<tr>
<td>Predicate Registers</td>
<td>64</td>
</tr>
<tr>
<td>Branch Registers</td>
<td>8</td>
</tr>
<tr>
<td>Application Registers</td>
<td>128</td>
</tr>
<tr>
<td>NN CPUID Registers</td>
<td></td>
</tr>
<tr>
<td>Instruction Pointer</td>
<td></td>
</tr>
<tr>
<td>User Mask</td>
<td></td>
</tr>
<tr>
<td>Current Frame Marker</td>
<td></td>
</tr>
<tr>
<td>NN Perf. Mon. Data Reg’s</td>
<td></td>
</tr>
</tbody>
</table>
IA64 Registers

- Integer registers
  - 128 in total; Width is 64-bits + 1 bit (NaT); r0 = 0
  - Integer, Logical and Multimedia data

- Floating point registers
  - 128 in total; 82-bits wide
  - 17-bit exponent, 64-bit significand
  - f0 = 0.0; f1 = 1.0
  - Significand also used for two SIMD floats

- Predicate registers
  - 64 in total; 1-bit each (fire/ do not fire)
  - p0 = 1 (default value)

- Branch registers
  - 8 in total; 64-bits wide (for address)
Floating-Point Load/Store

In matrix form:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Ldf.</th>
<th>Ldfp.</th>
<th>Stf.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>s</td>
<td>s</td>
<td>s</td>
</tr>
<tr>
<td>Double</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>Integer</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Dbl.Ext.</td>
<td>e</td>
<td>-</td>
<td>e</td>
</tr>
<tr>
<td>82-bits</td>
<td>fill</td>
<td>-</td>
<td>spill</td>
</tr>
<tr>
<td>Post-incr.</td>
<td>Reg/ Imm</td>
<td>8/ 16</td>
<td>Imm</td>
</tr>
</tbody>
</table>
IEEE 754 format

Intrinsic construct

- Sign/ Unsigned Exponent/ Unsigned Significand
  - \((-1)^S \times 2^E \times 1.f\)  
  - Example: \(-3 = (-1)^1 \times 2^1 \times 1.5\)
  - A fixed bias is added to the exponent: \(E' = E + b\)
  - Only the fractional part of significand is stored
    - Normalisation enforces “1.”

How is it stored:

- Single precision: \(1 + 8 + 23\) bits
- Double precision: \(1 + 11 + 52\) bits

In IA64 registers:

- Double Extended: \(1 + 17 + 64\) bits
  - Significand in register includes “1.”
    - This allows unnormalised numbers to be used as well
**Exponent representation**

- **In general:**
  - N bits allow 0 - (2^N-1)
  - Bias is defined as: 2^{N-1}-1
  - Exponent of 0: 0
  - Lowest ‘normal’ exp.: 1
    - Equivalent to 2^{-(2^{N-1}-2)}
  - Exponent of 1: 2^{N-1}-1
  - Highest ‘normal’ exp.: 2^N-2
    - Equivalent to 2^{(2^{N-1}-1)}
  - Infinity and NaNs: 2^N-1

- **Single Precision:**
  - 8 bits allow 0 - 255
    - 127
    - 0
    - 1
      - Equivalent to 2^{-126}
    - 127
    - 254
      - Equivalent to 2^{127}
    - 255
IA64 number range

- **Single:**
  - Range of \([2^{-126}, 2^{127}]\) corresponds to about \([10^{-37.9}, 10^{38.2}]\)
  - 23-bit accuracy: \(\sim 10^{-6.9}\)

- **Double:**
  - Range of \([2^{-1022}, 2^{1023}]\) corresponds to about \([10^{-307.7}, 10^{308.0}]\)
  - 52-bit accuracy: \(\sim 10^{-15.7}\)

- **Double Extended:**
  - Range of \([2^{-16382}, 2^{16383}]\) corresponds to about \([10^{-4931.5}, 10^{4931.8}]\)
  - 63-bit accuracy: \(\sim 10^{-19.0}\)

- **Register format**
  - Range of \([2^{-65535}, 2^{65536}]\) corresponds to about \([10^{-19728.0}, 10^{19728.3}]\)
  - 63-bit accuracy: \(\sim 10^{-19.0}\)
FLP Status Register

More on Traps

- Included in global FPSR
  - Inexact/ underflow/ overflow/ zero-
    divide/ denorm/ invalid ops.
  - Disable trap by setting corresponding flag

Status Fields
  - In an individual Status Field, the Trap Control bit can be set
**FLP Status Register**

- **Four Status Fields**
  - Sf0 (main status field), sf1, sf2, sf3

- **Flags**
  - Inexact, Underflow, Overflow, Zero Divide
  - Denorm/ Unnorm Operand
  - Invalid Operation

- **Contains Controls**
  - Trap Disabling
  - Rounding Control
  - Precision Control
  - Widest-range-exponent, Flush-to-zero
Floating-Point Operations

- **Standard instruction:**
  - \((qp)\) ops.pc.sf \(f_1 = f_3, f_4, f_2\)

- **Valid Operations:**
  - **Fma** \([U = X \times Y + Z]\)
  - **Fms** \([U = X \times Y - Z]\)
  - **Fnma** \([U = -(X \times Y) + Z]\)

\[U = X \times Y\]  \(fmul\)  Pseudo-op  With \(f0 = 0.0\)
\[U = X + Z\]  \(fadd\)  Pseudo-op  With \(f1 = 1.0\)
\[U = X - Z\]  \(fsub\)  Pseudo-op  With \(f1 = 1.0\)
SI MD Floating-Point

- **Standard instruction:**
  
  \[(qp) \text{ ops.pc.sf} \quad f_1 = f_3, f_4, f_2\]

- **Valid Operations:**
  
  - \(\text{Fpma } [U = X \times Y + Z]\)
  
  - \(\text{Fpms } [U = X \times Y - Z]\)
  
  - \(\text{Fpnma } [U = -(X \times Y) + Z]\)

NB: \(f_1\) does NOT contain two 32-bit versions of 1.0
Arithmetic Instructions

- Both for Normal and Parallel representation:
  - Multiply and Add \([f(p)ma]\)
  - Multiply and Subtract
  - Negate Multiply and Add
  - Reciprocal Approximation \([f(p)rcpa]\)
  - Reciprocal Square Root Approximation \([f(p)rsqrta]\)
  - Compare \([f(p)cmp]\)
  - Minimum \([f(p)\text{min}]\), Maximum \([f(p)\text{max}]\)
  - Absolute Minimum \([f(p)\text{amin}]\)
  - Absolute Maximum \([f(p)\text{amax}]\)
  - Convert to Signed/ Unsigned Integer \([f(p)cvt.fx(u)]\)

- Normal only:
  - Convert from Signed Integer \([fcvt.xf]\)
  - Integer Multiply and Add \([xma]\)
Non-arithmetic Instructions

- Both for Normal and Parallel representation:
  - Merge \([f(p)\text{merge}]\)
  - Classify \([f\text{class}]\)

- Parallel only:
  - Mix Left/ Right
  - Sign-Extend Left/ Right
  - Pack
  - Swap
  - And
  - Or
  - Select
  - Exclusive Or \([fxor]\)

- Status Control:
  - Check Flags
  - Clear Flags
  - Set Controls
Divide Example

- How do we achieve an accurate result \((x/y)\)?
  - \(\text{Frcpa only ‘guarantees’ 8.68 bits}\)
  - \(Z = x/y = x/(y’) \times x/(1 - d)\)
  - Implying: \(y = (y’)(1 - d)\) \(d = 1 - y \times rcp\), when \(rcp = 1/(y’)\)
  - Use polynomial expansion of \(1/(1-d) = 1 + d + d^2 + d^3 + \ldots\)
    - Rearranged: \((1 + d)(1+ d^2)(1+ d^4)(1+ d^8)\ldots\)
  - Precision doubles: 8.7 17.3 34.6 69.4 138.7
- Full formula:
  - \(rcp = 1/y\)
  - \(d = 1.0 - y \times rcp\)
  - \(rcp = rcp \times (1 + d)(1+ d^2)(1+ d^4)\)
  - \(z_0 = \text{double}(x \times rcp)\)
  - \(rem = x - z^*y\) \hspace{1cm} // remainder
  - \(z = \text{double}(z_0 + rem*rcp)\)
- Cost:
  - 10 operations (8 groups)
FLP Divide

Actual code:

```
define divide:
    frcpa.s0 f6,p2=f5,f4  // rcp = 1.0/y
    ;;
    fnma.s1 f7=f6,f4,f1   // d1 = - y * rcp + 1.0
    ;;
    fma.s1 f6=f7,f6,f6    // rcp = rcp (1.0 + d1)
    (p2)
    fmpy.s1 f9=f7,f7      // d2 = d1 * d1
    ;;
    fma.s1 f6=f9,f6,f6    // rcp = rcp * (1.0 + d2)
    (p2)
    fmpy.s1 f10=f9,f9     // d4 = d2 * d2
    ;;
    fma.s1 f6=f10,f6,f6   // rcp = rcp * (1.0 + d4)
    ;;
    fmpy.d.s1 f8=f5,f6    // z0 = x * rcp
    ;;
    fnma.s1 f11=f8,f5,f4  // rem = - y * rcp + x
    ;;
    fma.d.s0 f8=f8,f6,f11 // z = z + rem * rcp
```
**Integer divide**

- **Steps needed:**
  - Transfer variables
  - Convert to FLP
  - Perform the Division
  - Convert to integer
  - Transfer back

- **Issue:**
  - Long latency

```c
idiv:
  setf.sig f4=r4 // a
  setf.sig f5=r5 // b

;;
  fcvt.xf f4=f4 // convert to floating
  fcvt.xf f5=f5 //

;;
  do_div f4,f5 // precision dependent

;;
  fcvt.fx.trunc.s1 f8=f8 // convert to integer

;;
  getf.sig r8=f8 // c =a/ b
```

What if we need just the remainder?

Macro as already shown

11 June 1999
Steps needed:
- Transfer variables
- Convert to FLP
- Do the Division
- Compute remainder
- Convert to integer
- Transfer back

Issue:
- Even longer latency

irem:
```plaintext
setf.sig f4=r4  // a
setf.sig f5=r5  // b

;,
fcvt.xf f4=f4  // convert to floating
fcvt.xf f5=f5  //

;,
do_div f4,f5  // precision dependent

;,
fnma f6=f5,f8,f4  // quotient in f8

;,
fcvt.fx.trunc.s1 f6=f6  // convert to integer

;,
getf.sig r6=f6  // remainder
```

Macro as already shown
Integer multiply and add

Native instruction

- Running on the FLP side
  - (qp) xma.comp  \( f_1 = f_3, f_4, f_2 \)

- Valid completers:
  - Low (& low unsigned): l
  - High: h
  - High unsigned: hu

```
imul:
    setf.sig  f2=r2  // move from int
    setf.sig  f3=r3  // move from int

    xma.l  f8=f2,f3,f0  // result of mul in f8

    getf.sig  r8=f8  // return to integer
```
Part 4

Optimisation
Optimisation Strategy

As I see it:

- Work on the overall design
  - Control flow
  - Data flow

- Use optimal algorithms
  - In each important piece of code

- At the assembly level
  - Must have good architectural knowledge
  - Understand the chip implementation
  - Maybe use of special “tricks”

- C/ C++
  - Verify that compiler output is (at least) reasonable
  - Possibly, use inline assembler
Loops in assembly

Exploit (in priority order)

- Architectural support
  - Modulo Scheduling support
    - Predication
    - Register Rotation (Large Register Files)
  - Full access to other features
    - SIMD, Prefetching, Load pair instructions, etc.

- Chip implementation
  - Number of parallel slots; Execution units; Latencies
  - Cache sizes, Bandwidth

- Tricks
  - For increased speed
    - integer multiplication via shladd-sequences, etc.
  - For balanced execution capability (FLP INT)
“What do you get thanked for”

- Understand the hardware architecture
  - In order to make changes that matter
  - Some examples:
    - Integer registers:
      - Minimised use of allocated set (on the stack)
    - Control floating-point registers:
      - 1) No use  2) Use of fixed set  3) Use of total set
    - Prefetching
      - Use “nta” if you do not need the data again
The rotating integer registers serve as a stack

- Each routine allocates via ”Alloc” instruction:
  - Input + Local + Output
  - “Input + Local” may rotate (in sets of 8 registers)

- Proc A
  - Local A
  - Output A

- Proc B
  - Local B
  - Output B

- Proc C

- Proc B

- Proc A
  - Local A
  - Output A

Further Calls
IA-64 Speed Goal

- Fill each bundle ENTIRELY

- Two “easy” cases
  - 1) Initialisation
    - A lot of unrelated stuff can be packed together
  - 2) Loops
    - See section on Software Pipelining later on

- One “difficult” case:
  - Only ONE algorithm with LITTLE or NO inherent parallelism

\[
\begin{align*}
R &= T + \ldots \\
S &= R \times \ldots \\
X &= S - \ldots \\
Y &= X/\ldots \\
Z &= Y + 
\end{align*}
\]
Initial Example

- Look in detail at bundles
  - From two viewpoints
    - Fill the slots densely
    - Respect dependencies

getval:

```plaintext
alloc r3=ar.pfs,R_input,R_local,R_output,R_input+R_local
(p0) movl r2=Table
// No stop bit here
(p0) and r32=7,r32 // Choice is 0 - 7
// Embedded stop bit here
(p0) shladd r2=r32,4,r2 // Index table
;;
(p0) ldf.fill f8=[r2] // Load value
(p0) mov ar.fps=r3
(p0) br.ret.sptk.few b0 // return
```

Explicit Stop bit
Or Enforced Bundle Break

MLX
M+MI
MI B

3 groups in 3 bundles

11 June 1999
Parallel Compares

- Instruction format:
  - (qp) cmp.crel.cctype $p_1, p_2 = r_2, r_3$
  - (qp) cmp.crel.cctype $p_1, p_2 = \text{Imm}_8, r_3$
  - (qp) cmp.crel.cctype $p_1, p_2 = r_0, r_3$

- In the first two cases:
  - Only ‘eq’ (or ‘ne’) relationship may be used

- In the third case:
  - Can use ‘lt’ (or a variant) together with $r_0$
Use Parallel Compare

- If \((a || b || c || d)\) \{ ... \}

  - Serially:
    - (p0) cmp.ne.unc p_yes,p0=a,0 ;
    - (p0) cmp.ne p_yes,p0=b,0 ;
    - (p0) cmp.ne p_yes,p0=c,0 ;
    - (p0) cmp.ne p_yes,p0=d,0 ;

  - Parallel:
    - (p0) cmp.ne.unc p_yes,p0=a,0 ;
    - (p0) cmp.ne.or p_yes,p0=b,0
    - (p0) cmp.ne.or p_yes,p0=c,0
    - (p0) cmp.ne.or p_yes,p0=d,0 ;

Any one (of the three) may write a “1” into p_yes

Another variant would be to code all four compares in the same group; provided that a prior instruction has initialised p_yes to 0

11 June 1999
Line prefetch

- Place a cache-line at a given level
  - (qp) lfetch.lftype.lfhint [r₃], r₂
  - (qp) lfetch.lftype.lfhint [r₃], Imm₉

- Types are:
  - None
  - Fault

- Hints are:
  - None, nt1, nt2, nta
    - Non-temporal L1, L2, All levels
Load hints

- Decide where to place a line in cache

<table>
<thead>
<tr>
<th>Registers</th>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TS</td>
<td>TS</td>
<td>TS</td>
</tr>
<tr>
<td></td>
<td>NTS</td>
<td>NTS</td>
<td>NTS</td>
</tr>
<tr>
<td>None (all)</td>
<td>NT1 (Lfetch/Id)</td>
<td>NT2 (Lfetch)</td>
<td>NTA (all)</td>
</tr>
</tbody>
</table>
Modulo Scheduled Loop

Example:

- Copy integer data inside cache
  - 128 words (8B each)

- Use modulo scheduled loop (software pipelining)
  - Set Loop Count/ Epilogue Count
  - Assume all data in L0 cache
  - Hypothetical load access time with 3 delay cycles
Rotating Registers

- **Upper 75% rotate (when activated):**
  - General registers (r32-r127)
  - Floating Point Registers (f32-f127)
  - Predicate Registers (p16-p63)

- **Formula:**
  - Virtual Register = Physical Register - Register Rotation Base (RRB)
- Graphical representation
  - 7 loop traversals desired
  - Skewed execution
    - Stage 2 relative to Stage 1
    - Stage 3 relative to Stage 2
Modulo Loop - 3

How is it programmed?

By using:

- Rotating registers (Let values live longer)
- Predication
  - Each stage uses a distinct predicate register starting from p16
    - Stage 1 controlled by p16
    - Stage 2 by p17
    - Etc.
- Architected loop control using BR.CTOP
  - Clock down LC & EC
  - Set \( p16 = 1 \) when \( LC > 0 \)
    - [Actually p63 before new rotation]
  - Set \( P16 = 0 \) otherwise
Rotating Registers

Reminder of basic principle

- Just like “ageing”
- Virtual Register Number increases by 1 at the bottom of the loop:
  - r32 r33 r34 r35 (p16 p17 p18, and so on)
- Data is retained
  - Unless a new assignment is made
Modulo Loop - 5

- **Putting together the loop**
  - In a single bundle
    - With Store instruction that starts 3 cycles after the Load
    - Stage 1: ld8
    - Stage 2, Stage 3 (empty)
    - Stage 4: st8

```
move ar.lc=127
move ar.ec=4
move pr.rot=0x10000 // Initialise p16
;;
loop:
  ld8  r32=[ra],8   // Load value
  st8  [rb]=r35,8  // Store value
  br.ctop.sptk.few loop // Loop
;;
```
Which loops?

- Only the innermost loop
  - In this example,
    - L3 can be a Modulo Loop

- What if
  - L2 is the time-consuming loop?

- Several options to ensure good Modulo Scheduling
  1) Unroll the loop L3 completely
  2) Invert the loops
  3) Condense the loops
  4) Move L3 outside L2
    - Leaving just a predicated branch
    - And jump to it (when needed)
  5) Leave it in place
    - And manage it yourself
## Appendix 1a

### A-Class Instructions

- Whole set
  - Integer ALU
  - Compare
  - Multimedia ALU

<table>
<thead>
<tr>
<th>Type</th>
<th>Instructions</th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Add/ Sub (Register) And/ Andcm/ Or/ Xor</td>
<td>Integer</td>
</tr>
<tr>
<td>A2</td>
<td>Shladd</td>
<td>I</td>
</tr>
<tr>
<td>A3</td>
<td>Sub (Immediate) And/ Andcm/ Or/ Xor</td>
<td>I</td>
</tr>
<tr>
<td>A4</td>
<td>Adds</td>
<td>I</td>
</tr>
<tr>
<td>A5</td>
<td>Addl</td>
<td>I</td>
</tr>
<tr>
<td>A6</td>
<td>Compare (Reg.)</td>
<td>I</td>
</tr>
<tr>
<td>A7</td>
<td>Compare to Zero</td>
<td>I</td>
</tr>
<tr>
<td>A8</td>
<td>Compare (Imm.)</td>
<td>I</td>
</tr>
<tr>
<td>A9</td>
<td>Padd/ Psub/ Pavg/ Pcmp</td>
<td>Multimedia</td>
</tr>
<tr>
<td>A10</td>
<td>Pshladd/ Pshradd</td>
<td>MM</td>
</tr>
</tbody>
</table>
Call to Action

- Study the Architecture Manual
  - Few items at a time
    - This is dense material

- Write code snippets:
  - Exercising the different architectural features
  - Compare to existing architectures (such as IA32)

- Be ready for the first shipments of hardware