IA-64 architecture

A Detailed Tutorial

Version 3

Sverre Jarp
CERN - IT Division

http://nicewww.cern.ch/~sverre/SJ.html
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

8 November 1999
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 (a la 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:
  
  ```c
  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.pfs=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

Application registers

Register allocation

Enforced Bundle Break

Predicated execution

Branch return

```plaintext
.proc getval:
  alloc   r3=ar.pfs,R_input,R_local,R_output,R_input+R_local
(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.pfs=r3
(p0)  br.ret.sptk.few b0  // return
```
# User Register Overview

<table>
<thead>
<tr>
<th>Register Type</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 Registers</td>
<td></td>
</tr>
</tbody>
</table>

- Instruction Pointer
- User Mask
- Current Frame Marker
- NN Perf. Mon.
  - Data Reg’s
General information about the processor

- At least 5 registers:

<table>
<thead>
<tr>
<th>CPUID[0]</th>
<th>Vendor</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUID[1]</td>
<td>Name</td>
</tr>
<tr>
<td>CPUID[2]</td>
<td>Processor Serial Number</td>
</tr>
</tbody>
</table>
IA64 Common Registers

- **Integer registers**
  - 128 in total; Width is 64-bits + 1 bit (NaT); \( r_0 = 0 \)
  - Integer, Logical and Multimedia data

- **Floating point registers**
  - 128 in total; 82-bits wide
  - 17-bit exponent, 64-bit significand
  - \( f_0 = 0.0; f_1 = 1.0 \)
  - Significand also used for two SIMD floats

- **Predicate registers**
  - 64 in total; 1-bit each (fire/ do not fire)
  - \( p_0 = 1 \) (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
          ↓    ↓    ↓    ↓    ↓    ↓    ↓    ↓    ↓    ↓    ↓    ↓
```

8 November 1999
## Register Convention

**Run-time:**

- **Branch Registers:**
  - B0: Call register
  - B1-B5: Must be preserved
  - B6-B7: Scratch

- **General Registers:**
  - R1: GP (Global Data Pointer)
  - R2-R3: scratch
  - R4-R7: Must be preserved
  - R8-R11: Procedure Return Values
  - R12: Stack Pointer
  - R13: (Reserved as) Thread Pointer
  - R14-R31: Scratch
  - R32-Rxx: Argument Registers
Register Convention (2)

- **Run-time convention**
  - **Floating-Point:**
    - F2-F5: Preserved
    - F6-F7: Scratch
    - F8-F15: Argument/Return Registers
    - F16-F31: Must be preserved
    - F32-F127: Scratch
  
  - **Predicates:**
    - P1-P5: Must be preserved
    - P6-P15: Scratch
    - P16-P63: Must be preserved
  
  - **Additionally:**
    - Ar.unat & Ar.lc: Must be preserved
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

- Local A  Output A

Proc B

- Input B + Local B  Output B

Proc C

Proc B

Proc A

Local A  Output A

8 November 1999
Which registers to use

- **Start with alloc:**
  - Alloc r36=ar.pfs,4,4,2,8

- **Rotation should only be activated**
  - When input registers have been read

- **Lots of register below r32:**
  - r2-r3, r14-31 (scratch)
  - r8-r11 (return values; work registers before)
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 Instruction Slots
  - 5 bits for Template
    - Typical examples: MFI or MIB
    - Including bit for Bundle Break “S”

- A bundle of 16B:
  - Basic unit for expressing parallelism
  - The unit that the Instruction Pointer points to
  - The unit you branch to
  - Actually executed may be less, equal, or more

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

- 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)
- SIMD Floating Point operations (e.g. fpma)
- Memory operations
- Move operations
- Cache Management operations
Conventions

- Instruction syntax
  - \((qp) \ ops[.\comp_1] \quad r_1 = r_2, r_3\)
  - 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
  - \(A^o\) 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
Logical Operations

Instruction format:

- \((qp) \ ops \ r_1 = r_2, r_3\)
- \(Imm_8, r_3\)

Valid Operations:

- And
- Or
- Xor (Exclusive Or)
- Andcm (And Complement)
  - Result_1 = Input_2 \& \neg Input_3
Arithmetic Operations

Instruction format:

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

Valid Operations:

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

NB: Integer multiply is a FLP operation
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 \( p_1, p_2 = r0, r_3 \)

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) ops₁  \( r₁ = r₃, r₂ \)
- (qp) ops₁[.u]  \( r₁ = r₃, \text{count}_6 \)
- (qp) extr[.u]  \( r₁ = r₃, \text{pos}_6, \text{len}_6 \)
- (qp) dep[.z]  \( r₁ = r₂, r₃, \text{pos}_6, \text{len}_4 \)
- (qp) shrp[.u]  \( r₁ = r₃, r₂, \text{count}_6 \)

Valid Operations:

- ops₁ 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
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 \( r_1 = [r_3], \text{Imm} \)
  - (qp) ldf.fsz.fldtype.ldhint \( f_1 = [r_3], r_2 \)
  - (qp) ldf.fsz.fldtype.ldhint \( f_1 = [r_3], \text{Imm} \)

- **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], \text{Imm}_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]
Branch Operations

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

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

- Sequential \textbf{Prefetch Hint}
  - Few/none – few lines
  - Many

- Branch Cache \textbf{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 \(r_1 = r_2, count_2, r_3\)
  - **Shift** \(r_2\) **Left** by \(count_2\) and **ADD** to \(r_3\)

- Fnma.s1 \(f_1 = f_3, f_4, f_2\)
  - **Flp Negative Multiply** and **Add**: \(f_1 = - (f_3 \times f_4) + f_2\)

- Less Obvious is: Andcm
  - **AND Complement**: \(r_1 = \text{Input}_2 \& \sim\text{Input}_3\)
  - Complement \text{Input}_2 or \text{Input}_3??
Multimedia 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>CPUID D Registers</td>
<td></td>
</tr>
</tbody>
</table>

### Data Registers
- Instruction Pointer
- User Mask
- Current Frame Marker
- NN Perf. Mon. Data Reg’s
**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 a"o 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>1</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>Pshladd</td>
<td>-</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>Pshradd</td>
<td>-</td>
<td>2</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></th>
<th>1B</th>
<th>2B</th>
<th>4B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pshl/ Pshr</td>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Pshr.u</td>
<td>-</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

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

- \((qp)\) `pmpy2.r`` \(r_1 = r_2, r_3``
- Same instruction for left

Parallel Multiply and Shift Right

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

Intermediate Results

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**
  - \((qp)\) \(psad1 \ r_1 = r_2, r_3\)
  - 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 pieces

Example 1: Unpack1.h

Example 2: Mix1.l

Both are 12

8 November 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
Complex Multimedia - 5

- **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 \( 0x80 \) <-> \( 0x7F \) \( [0x8000 \) <-> \( 0x7FFF] \)
  - “unsigned” covers \( 0x00 \) <-> \( 0xFF \) \( [0x0000 \) <-> \( 0xFFFF] \)

Parallel add/subtract

- (qp) padd4 \( r_1 = r_2, r_3 \)
  - Modulo arithmetic
Parallel compare

- \((qp)\) \text{pcmpn.prel} \quad 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

<table>
<thead>
<tr>
<th>mux2</th>
<th>r34=r32,0x50</th>
</tr>
</thead>
<tbody>
<tr>
<td>r35</td>
<td>r33,0x14</td>
</tr>
<tr>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>pmpyshr2.u</td>
<td>r36=r34,r35,0</td>
</tr>
<tr>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>pmpyshr2.u</td>
<td>r37=r34,r35,16</td>
</tr>
<tr>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>mix2.r</td>
<td>r38=r37,r36</td>
</tr>
<tr>
<td>mix2.l</td>
<td>r39=r37,r36</td>
</tr>
<tr>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>shr.u</td>
<td>r40=r39,32</td>
</tr>
<tr>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>zxt2</td>
<td>r41=r39</td>
</tr>
<tr>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>r42=r40,r41</td>
</tr>
<tr>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>shl</td>
<td>r43=r42,16</td>
</tr>
<tr>
<td>;;</td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>r31=r43,r38</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>A*h</td>
<td>a*h</td>
</tr>
<tr>
<td>b*h</td>
<td>a*b</td>
</tr>
<tr>
<td>A*h</td>
<td>a*h</td>
</tr>
<tr>
<td>b*h</td>
<td>a*b</td>
</tr>
<tr>
<td>a*h</td>
<td>a*b</td>
</tr>
<tr>
<td>a*h</td>
<td>a*b</td>
</tr>
<tr>
<td>a*b</td>
<td>a*b</td>
</tr>
<tr>
<td>a*b</td>
<td>a*b</td>
</tr>
</tbody>
</table>

Contributed by Walter Misar (TU - Darmstadt)
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]
psadbw mm2,[edi+8]
add esi,eax  // increment pointer
add edi, eax
paddw mm0, mm1  // accumulate
paddw mm7, mm2
dec ecx
jp 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 ;;
psad1 r32=r32,r34
psad1 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

- 128 Integer Registers
- 128 Floating Point Registers
- 64 Predicate Registers
- 8 Branch Registers
- 128 Application Registers
- NN CPUID Registers
- Instruction Pointer
- User Mask
- Current Frame Marker
- NN Perf. Mon. Data Reg’s
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 Loads/Stores

#### 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: \(\sim 10^{-19.0}\)
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
Four Status Fields

- Sf0 (main status field), sf1, sf2, sf3

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

- Contains Control
  - Trap Disabling
  - Rounding Control
  - Precision Control
  - Widest-range-exponent, Flush-to-zero

<table>
<thead>
<tr>
<th>flags</th>
<th>control</th>
</tr>
</thead>
<tbody>
<tr>
<td>i u o z d v</td>
<td>td rc pc w f</td>
</tr>
</tbody>
</table>
Floating-Point Operations

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

- **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 $f1 = 1.0$
SIMD 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 [\( U = - (X \times Y) + Z \)]

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 \([f(p)min]\), Maximum \([f(p)max]\)
  - Absolute Minimum \([f(p)amin]\)
  - Absolute Maximum \([f(p)amax]\)
  - 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 \([f\text{xor}]\)

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

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

- Cost:
  - 10 operations (8 groups)
## FLP Divide

### Actual code:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>divide:</td>
<td></td>
</tr>
<tr>
<td>frcpa.s0 f6,p2=f5,f4</td>
<td>// rcp = 1.0/ y</td>
</tr>
<tr>
<td>fnma.s1 f7=f6,f4,f1</td>
<td>// d1 = - y * rcp + 1.0</td>
</tr>
<tr>
<td>fma.s1 f6=f7,f6,f6</td>
<td>// rcp = rcp (1.0 + d1)</td>
</tr>
<tr>
<td>fmpy.s1 f9=f7,f7</td>
<td>// d2 = d1 * d1</td>
</tr>
<tr>
<td>fma.s1 f6=f9,f6,f6</td>
<td>// rcp = rcp * (1.0 + d2)</td>
</tr>
<tr>
<td>fmpy.s1 f10=f9,f9</td>
<td>// d4 = d2 * d2</td>
</tr>
<tr>
<td>fma.s1 f6=f10,f6,f6</td>
<td>// rcp = rcp * (1.0 + d4)</td>
</tr>
<tr>
<td>fmpy.d.s1 f8=f5,f6</td>
<td>// z0 = x * rcp</td>
</tr>
<tr>
<td>fnma.s1 f11=f8,f5,f4</td>
<td>// rem = - y * rcp + x</td>
</tr>
<tr>
<td>fma.d.s0 f8=f8,f6,f11</td>
<td>// z = z + rem * rcp</td>
</tr>
</tbody>
</table>
Steps needed:
- Transfer variables
- Convert to FLP
- Perform the Division
- Convert to integer
- Transfer back

Issue:
- Long latency

What if we need just the remainder?

```
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
```
Steps needed:

- Transfer variables
- Convert to FLP
- Do the Division
- Compute remainder
- Convert to integer
- Transfer back

Issue:

- Even longer latency

```c
irem: 
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
;
fsvt.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
```
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.
  - **Micro-architecture**
    - 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)

8 November 1999
“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
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
Local A  Output A

Proc B
Local B  Output B

Proc C

Proc B

Proc A
Local A  Output A

Further Calls
```

8 November 1999
Execution Width

- A given implementation could be N wide
  - Itanium/ Merced is implemented as a “two-banger”
    - 6 parallel instructions
      - Major enhancement compared to IA-32
    - But,
      - If nothing useful is put into the syllables, they get filled as NOPs

This template should be even (i.e. without stop bit)
**Instruction Delivery**

- **Must match**
  - instructions to issue ports
    - w/ corresponding execution units attached

![Dispersal network diagram](image)

- **9 available ports in total**
IA-64 Secret of Speed

- Fill the ENTIRE execution width

- 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
  - Example: RC6 (encryption)

\[
R = T + \ldots \\
S = R \ast \ldots \\
X = S - \ldots \\
Y = X / \ldots \\
Z = Y + \ldots
\]
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.pfs=r3
(p0)    br.ret.sptk.few b0 // return
```

3 groups in 3 cycles
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 r0
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
Line prefetch

- Place a cache-line at a given level
  - (qp) lfetch.lftype.lfhint \([r_3], r_2\)
  - (qp) lfetch.lftype.lfhint \([r_3], 1mm_9\)

- 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

- Registers
  - Level 1
    - TS
    - NTS
  - Level 2
    - TS
    - NTS
  - Level 3
    - TS
    - NTS

None (all)
- NT1
  - (Lfetch/Id)
- NT2
  - (Lfetch)
- NTA (all)
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)

\[
\begin{align*}
\text{f28} & \rightarrow \text{f29} & \rightarrow \text{f30} & \rightarrow \text{f31} & \rightarrow \text{f32} & \rightarrow \text{f33} & \rightarrow \text{f34} & \rightarrow \text{f35} & \rightarrow \ldots \ldots \\
\text{f124} & \rightarrow \text{f125} & \rightarrow \text{f126} & \rightarrow \text{f127} & \rightarrow \ldots \ldots
\end{align*}
\]
Modulo Loop - 2

- **Graphical representation**
  - 7 loop traversa” desired
  - Skewed execution
    - Stage 2 relative to Stage 1
    - Stage 3 relative to Stage 2
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
Modulo Loop - 4

- 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
Putting together the loop

- In a single bundle
  - With Store instruction that starts 3 cycles after the Load
  - Stage 1: ld8
  - Stage2, Stage 3 (empty)
  - Stage 4: st8

```plaintext
mov    ar.lc=127
mov    ar.ec=4
mov    pr.rot=0x10000  // Initialise p16

;;
loop:
(p16) ld8    r32=[ra],8  // Load value
(p19) 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
Action Call

- Study the Architecture Manual (and other available documents)
  - 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
## Appendix 1a

### A-Class Instructions

- Whole set
  - Integer ALU
  - Compare
  - Multimedia ALU

<table>
<thead>
<tr>
<th>Type</th>
<th>Instructions</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Add; Sub (Register) And; Andcm; Or; Xor</td>
<td>Integer ALU</td>
</tr>
<tr>
<td>A2</td>
<td>Shladd</td>
<td></td>
</tr>
<tr>
<td>A3</td>
<td>Sub (Immediate) And; Andcm; Or; Xor</td>
<td></td>
</tr>
<tr>
<td>A4</td>
<td>Adds</td>
<td></td>
</tr>
<tr>
<td>A5</td>
<td>Addl</td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td>Compare (Reg.)</td>
<td>Int. Compare</td>
</tr>
<tr>
<td>A7</td>
<td>Compare to Zero</td>
<td></td>
</tr>
<tr>
<td>A8</td>
<td>Compare (Imm.)</td>
<td></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></td>
</tr>
</tbody>
</table>
# Appendix 1b

- **L-instructions**
  - **Part 1**
    - Multimedia and Variable Shifts
    - Integer Shifts

<table>
<thead>
<tr>
<th>Type</th>
<th>Instructions</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>Pmpyshr</td>
<td>Multimedia</td>
</tr>
<tr>
<td>I2</td>
<td>Pmpy; Mix; Pack; Unpack; Pmin; Pmax; Psad</td>
<td>&quot;</td>
</tr>
<tr>
<td>I3</td>
<td>Mux1</td>
<td>&quot;</td>
</tr>
<tr>
<td>I4</td>
<td>Mux2</td>
<td>&quot;</td>
</tr>
<tr>
<td>I5</td>
<td>Shr; Pshr (Variable)</td>
<td>&quot;</td>
</tr>
<tr>
<td>I6</td>
<td>Pshr (Fixed)</td>
<td>&quot;</td>
</tr>
<tr>
<td>I7</td>
<td>Shl; Pshl (Variable)</td>
<td>&quot;</td>
</tr>
<tr>
<td>I8</td>
<td>Pshl (Fixed)</td>
<td>&quot;</td>
</tr>
<tr>
<td>I9</td>
<td>Population Count</td>
<td>&quot;</td>
</tr>
<tr>
<td>I10</td>
<td>Shrp</td>
<td>Int. Shift</td>
</tr>
<tr>
<td>I11</td>
<td>Extract</td>
<td>&quot;</td>
</tr>
<tr>
<td>I12</td>
<td>Zero and deposit</td>
<td>&quot;</td>
</tr>
<tr>
<td>I13</td>
<td>Zero and deposit (Imm.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>I14</td>
<td>Deposit (Imm.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>I15</td>
<td>Deposit</td>
<td>&quot;</td>
</tr>
</tbody>
</table>
## I-instructions

### Part 2

- Miscellaneous

<table>
<thead>
<tr>
<th>Type</th>
<th>Instructions</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>I16</td>
<td>Test Bit</td>
<td>Test Bit</td>
</tr>
<tr>
<td>I17</td>
<td>Test Nat</td>
<td>&quot;</td>
</tr>
<tr>
<td>I18</td>
<td>Move Long</td>
<td>Int. Misc.</td>
</tr>
<tr>
<td>I19</td>
<td>Break.i; Nop.i</td>
<td>&quot;</td>
</tr>
<tr>
<td>I20</td>
<td>Chk.s.i</td>
<td>&quot;</td>
</tr>
<tr>
<td>I21</td>
<td>Move to BR</td>
<td>Int. Move</td>
</tr>
<tr>
<td>I22</td>
<td>Move from BR</td>
<td>&quot;</td>
</tr>
<tr>
<td>I23</td>
<td>Move to Predicate (Reg.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>I24</td>
<td>Move to Predicate (Imm.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>I25</td>
<td>Move from PR/IP</td>
<td>&quot;</td>
</tr>
<tr>
<td>I26</td>
<td>Move to AR (Reg.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>I27</td>
<td>Move to AR (Imm.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>I28</td>
<td>Move from AR</td>
<td>&quot;</td>
</tr>
<tr>
<td>I29</td>
<td>Sign/ Zero Extend; Compute Zero Index</td>
<td>Int. Misc.</td>
</tr>
</tbody>
</table>
### M-instructions

- **Load**
- **Store**
- **Prefetch**

<table>
<thead>
<tr>
<th>Type</th>
<th>Instructions</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>Integer Load</td>
<td>Load/Store</td>
</tr>
<tr>
<td>M2</td>
<td>Integer Load (PI via reg.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>M3</td>
<td>Integer Load (PI via imm.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>M4</td>
<td>Integer Store</td>
<td>&quot;</td>
</tr>
<tr>
<td>M5</td>
<td>Integer Store (PI via imm.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>M6</td>
<td>Floating-Point Load</td>
<td>&quot;</td>
</tr>
<tr>
<td>M7</td>
<td>FLP Load (PI via reg.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>M8</td>
<td>FLP Load (PI via imm.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>M9</td>
<td>FLP Store</td>
<td>&quot;</td>
</tr>
<tr>
<td>M10</td>
<td>FLP Store (PI via imm.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>M11</td>
<td>FLP Load Pair</td>
<td>&quot;</td>
</tr>
<tr>
<td>M12</td>
<td>FLP Load Pair (PI via imm.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>M13</td>
<td>Line prefetch</td>
<td>Prefetch</td>
</tr>
<tr>
<td>M14</td>
<td>Line prefetch (PI via reg.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>M15</td>
<td>Line prefetch (PI via imm.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>Type</td>
<td>Instructions</td>
<td>Category</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>M16</td>
<td>(Cmp and) Exchange</td>
<td>Semaphore</td>
</tr>
<tr>
<td>M17</td>
<td>Fetch and Add</td>
<td>&quot;</td>
</tr>
<tr>
<td>M18</td>
<td>Setf</td>
<td>Set/ Get</td>
</tr>
<tr>
<td>M19</td>
<td>Getf</td>
<td>&quot;</td>
</tr>
<tr>
<td>M20</td>
<td>Chk.s.m (INT)</td>
<td>Speculation</td>
</tr>
<tr>
<td>M21</td>
<td>Chk.s (FLP)</td>
<td>&quot;</td>
</tr>
<tr>
<td>M22</td>
<td>Chk.a.nc/ clr (INT)</td>
<td>&quot;</td>
</tr>
<tr>
<td>M23</td>
<td>Chk.a.nc/ clr (FLP)</td>
<td>&quot;</td>
</tr>
<tr>
<td>M24</td>
<td>Sync; Fence; Serialize</td>
<td>Synchr.</td>
</tr>
<tr>
<td>M25</td>
<td>Flushrs</td>
<td>&quot;</td>
</tr>
<tr>
<td>M26</td>
<td>Invala.e (INT)</td>
<td>&quot;</td>
</tr>
<tr>
<td>M27</td>
<td>Invala.e (FLP)</td>
<td>&quot;</td>
</tr>
<tr>
<td>M28</td>
<td>Flush cache</td>
<td>&quot;</td>
</tr>
</tbody>
</table>
### M-instructions

- **Register moves**
- **Misc.**

<table>
<thead>
<tr>
<th>Type</th>
<th>Instructions</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>M29</td>
<td>Move to AR (Reg.)</td>
<td>Mem.Mov.</td>
</tr>
<tr>
<td>M30</td>
<td>Move to AR (Imm.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>M31</td>
<td>Move from AR</td>
<td>&quot;</td>
</tr>
<tr>
<td>M32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M34</td>
<td>Alloc</td>
<td>M.Misc.</td>
</tr>
<tr>
<td>M35</td>
<td>Move to PSR</td>
<td>&quot;</td>
</tr>
<tr>
<td>M36</td>
<td>Move from PSR</td>
<td>&quot;</td>
</tr>
<tr>
<td>M37</td>
<td>Break.m; Nop.m</td>
<td>&quot;</td>
</tr>
<tr>
<td>M38</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M39</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M41</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M44</td>
<td>Set/ Reset User Mask</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

8 November 1999
## B-instructions

- **Whole set**

<table>
<thead>
<tr>
<th>Type</th>
<th>Instructions</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>IP-relative branch</td>
<td>Branch</td>
</tr>
<tr>
<td>B2</td>
<td>IP-rel. Counted Branch</td>
<td>&quot;</td>
</tr>
<tr>
<td>B3</td>
<td>IP-rel. Call</td>
<td>&quot;</td>
</tr>
<tr>
<td>B4</td>
<td>Indirect Branch (B-reg.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>B5</td>
<td>Indirect Call (B-reg.)</td>
<td>&quot;</td>
</tr>
<tr>
<td>B6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B8</td>
<td>Clrrrb</td>
<td>Br.Misc.</td>
</tr>
<tr>
<td>B9</td>
<td>Break.b/ Nop.b</td>
<td>Br.Nop.</td>
</tr>
</tbody>
</table>
# Appendix 1h

## F-instructions

### Whole Set

- Arithmetic
- Compare and Classify
- Approximations
- Miscellaneous
- Convert
- Status Fields

<table>
<thead>
<tr>
<th>Type</th>
<th>Instructions</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>F(p)ma with variants</td>
<td>FLP Arith.</td>
</tr>
<tr>
<td>F2</td>
<td>Xma</td>
<td>&quot;</td>
</tr>
<tr>
<td>F3</td>
<td>Fselect</td>
<td>FLP Select</td>
</tr>
<tr>
<td>F4</td>
<td>Fcmp</td>
<td>FLP Compare</td>
</tr>
<tr>
<td>F5</td>
<td>Fclass</td>
<td>&quot;</td>
</tr>
<tr>
<td>F6</td>
<td>F(p)rcpa</td>
<td>FLP Approx.</td>
</tr>
<tr>
<td>F7</td>
<td>F(p)sqrta</td>
<td>&quot;</td>
</tr>
<tr>
<td>F8</td>
<td>F(p)min/ max; F(p)cmp</td>
<td>FLP Min/ Max</td>
</tr>
<tr>
<td>F9</td>
<td>F(p)merge + Logical</td>
<td>FLP M/ L</td>
</tr>
<tr>
<td>F10</td>
<td>Convert FLP to Fixed</td>
<td>FLP Convert</td>
</tr>
<tr>
<td>F11</td>
<td>Convert Fixed to FLP</td>
<td>&quot;</td>
</tr>
<tr>
<td>F12</td>
<td>Set Contro&quot;</td>
<td>FLP Status</td>
</tr>
<tr>
<td>F13</td>
<td>Clear Flags</td>
<td>&quot;</td>
</tr>
<tr>
<td>F14</td>
<td>Check Flags</td>
<td>&quot;</td>
</tr>
<tr>
<td>F15</td>
<td>Break.f/ Nop.f</td>
<td>FLP Misc.</td>
</tr>
</tbody>
</table>
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)

8 November:
- Version 3:
  - More editorial changes
  - Added slides on:
    - Register coding conventions
    - Itanium/ Merced execution width and units
    - Appendix w/ all instruction categories