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

Phase 1

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

Phase 2

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

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 EPIC:**
      - **“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:**

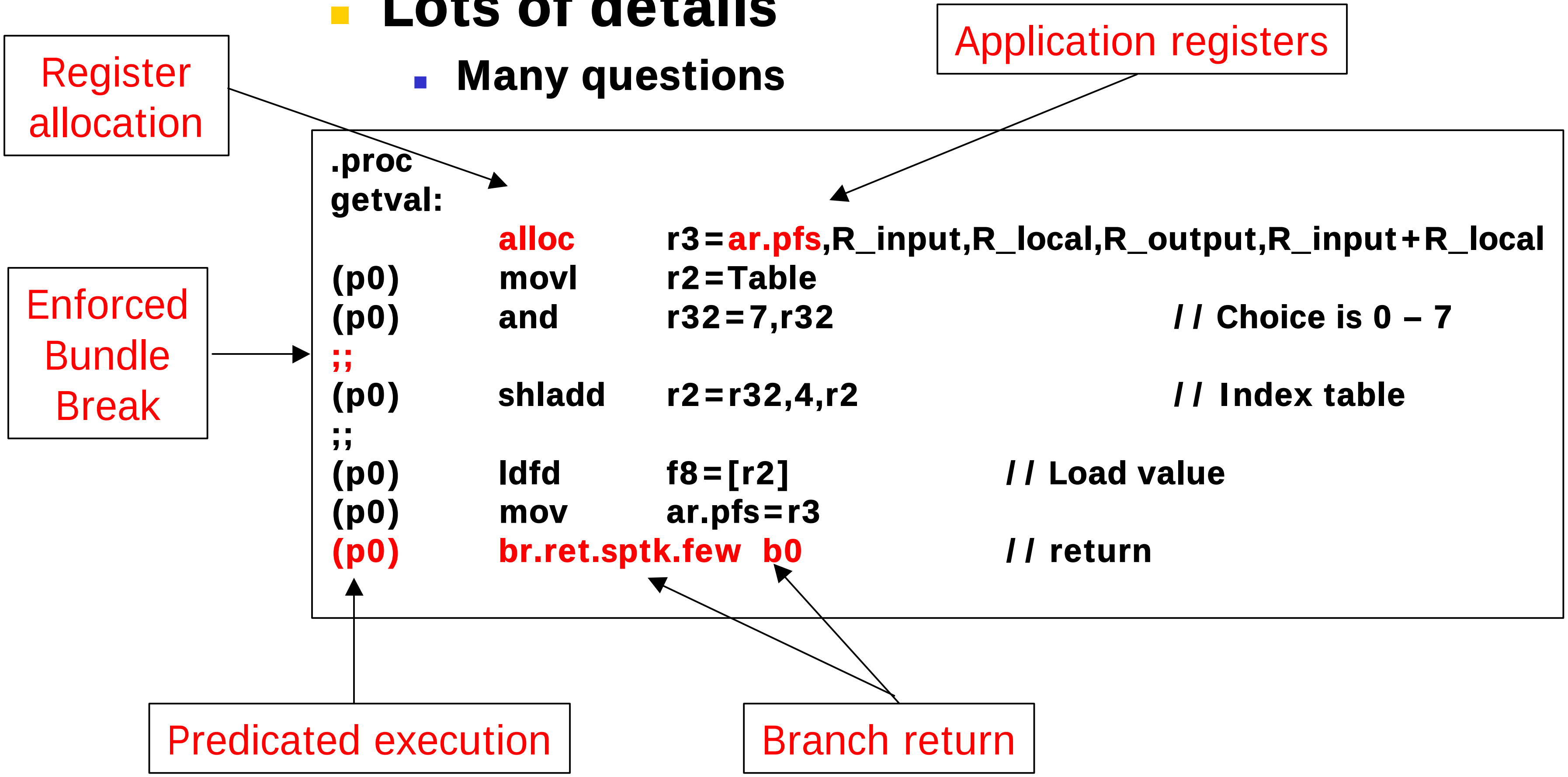
```
long Indx = 5   ;          // Choice may be 0 - 7
double My_fp = getval(Indx);
```

```
.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
Table:
          real8       5.99
          real8       ....
……
```

Not strictly needed for leaf routines

Initial explanation

- **Lots of details**
  - **Many questions**



Register allocation

Application registers

Enforced Bundle Break

```
.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
```

Predicated execution

Branch return

User Register Overview

128
Integer Registers

128
Floating Point Registers

64
Predicate Registers

8
Branch Registers

Instruction Pointer

User Mask

Current Frame Marker

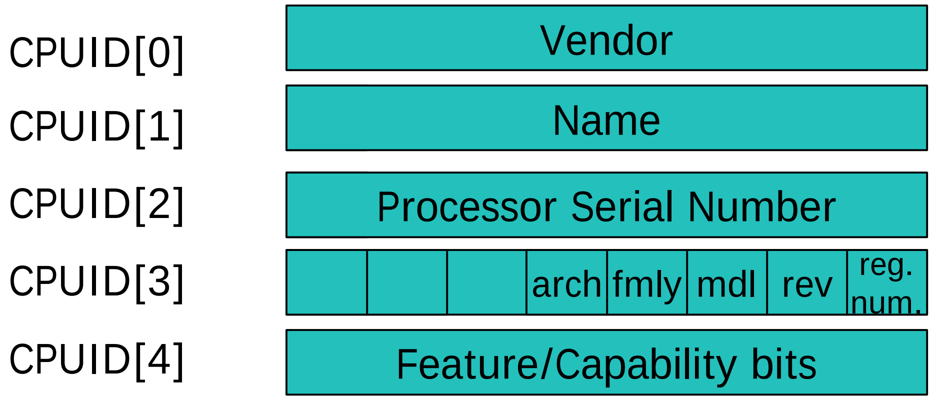
128
Application Registers

NN
CPUID Registers

NN Perf. Mon.
Data Reg’s

CPUID registers

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



CPUID[0] Vendor

CPUID[1] Name

CPUID[2] Processor Serial Number

CPUID[3] arch fmly mdl rev reg. num.

CPUID[4] Feature/Capability bits

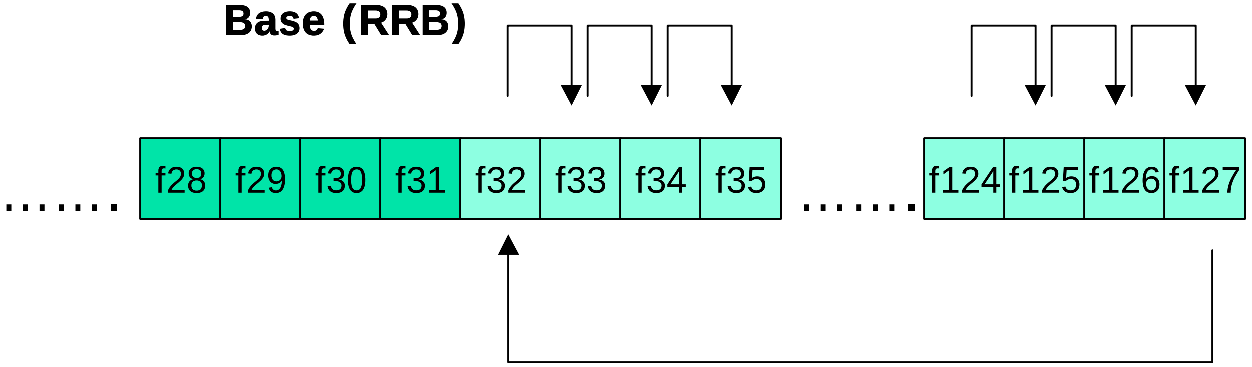
IA64 Common Registers

- **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)



Register Convention

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

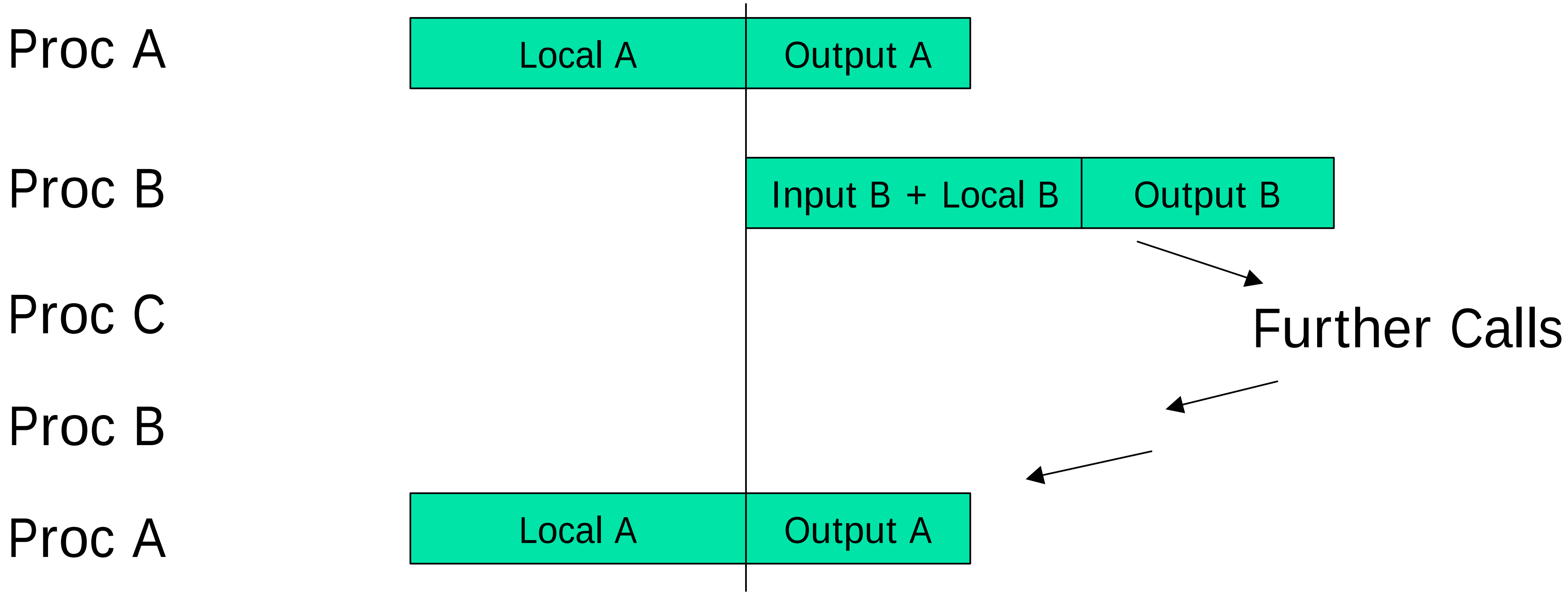
  - **General Registers:**
    - **R1: GP (Global Data Pointer)**
    - **R2-R3: scratch**
    - **R4-R7: Must be preserved**
    - **R8-R11: Procedure Return Values**
    - **R12: Stack Pointer**
    - **R13: (Reserved as) Thread Pointer**
    - **R14-R31: Scratch**
    - **R32-Rxx: Argument Registers**

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**

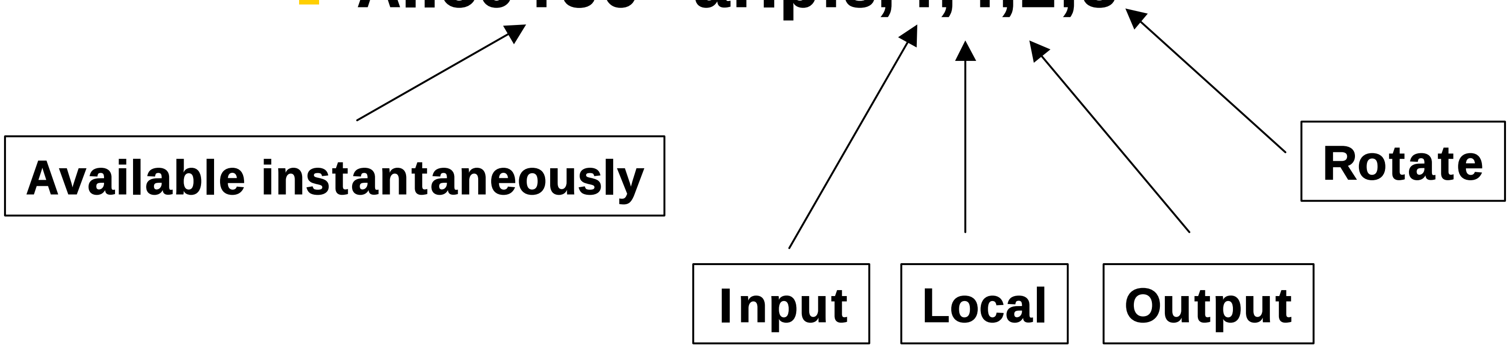
Register Stack

- **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)**



Which registers to use

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



Available instantaneously

Input

Local

Output

Rotate

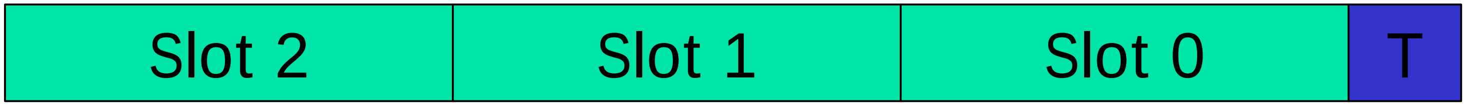
  - Rotation should only be activated
    - When input registers have been read
  - Lots of register below r32:
    - r2-r3, r14-31 (scratch)
    - r8-r11 (return values; work registers before)

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



Slot 2	Slot 1	Slot 0	T

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: MII, MI+I, MMI, MM+I, MFI
      - Ending in B: MIB, 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

FMA:

Opcode++	R4	R3	R2	R1	qp
7	7	7	7	7	6

Add:

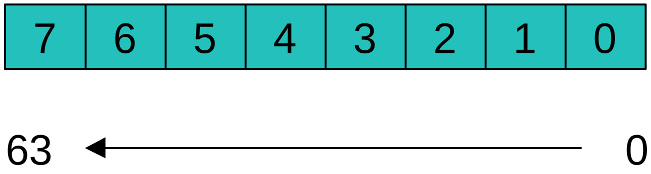
Opcode	Flags	R3	R2	R1	qp
7	7	7	7	7	6

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$_1$] $r_1 = r_2, r_3$**
    - **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**
  - **All right-to left**



- **Immediates**
  - **Various sizes exist**
  - **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 $r_1 = r_2, r_3$**
    - **(qp) ops $r_1 = Imm_8, r_3$**

  - **Valid Operations:**
    - **And**
    - **Or**
    - **Xor (Exclusive Or)**
    - **Andcm (And Complement)**
      - **$Result_1 = Input_2 \ \& \ {\sim}Input_3$**

Arithmetic Operations

- **Instruction format:**
  - **(qp) ops_1 $r_1 = r_2, r_3[,1]$**
  - **(qp) ops_2 $r_1 = Imm_x, r_3$**
  - **(qp) ops_3 $r_1 = r_2, count_2, r_3$**

**X86 Inc/Dec
replaced with
(qp) ops $r_1 = r_2$,r0,1**

**Z = Y – imm
becomes
(qp) Add r_1 =-imm, r_3**

- **Valid Operations:**
  - **Add**
  - **Sub**
  - **Adds/Addl (Imm_{14}, Imm_{22})**
  - **Shladd**

**Loading
an immediate value
(qp) Add r_1 =imm, r0**

- **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 = Imm_8, r_3
  - (qp) cmp.crel.ctype p_1, p_2 = r0, r_3

Parallel inequality form →

- **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

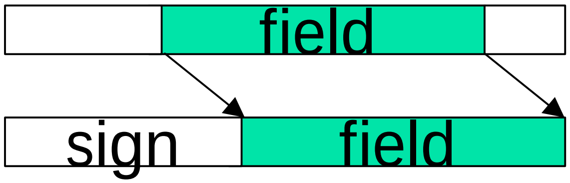
Shift Operations

- **Instruction format:**
  - (qp) ops$_1$ $r_1 = r_3, r_2$
  - (qp) ops$_1$[.u] $r_1 = r_3, count_6$
  - (qp) extr[.u] $r_1 = r_3, pos_6, len_6$
  - (qp) dep[.z] $r_1 = r_2, r_3, pos_6, len_4$
  - (qp) shrp[.u] $r_1 = r_3, r_2, count_6$

Deposit

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

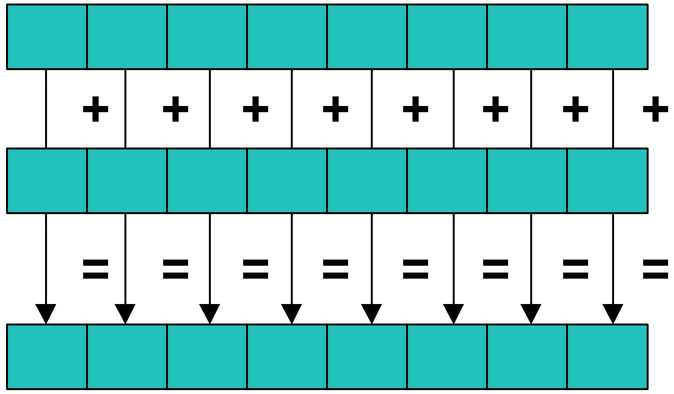
- **Valid Operations:**
  - ops$_1$ can be: Shl, shr, shr.u
- **Extract:**
  - Shift right and mask



Simple Multimedia

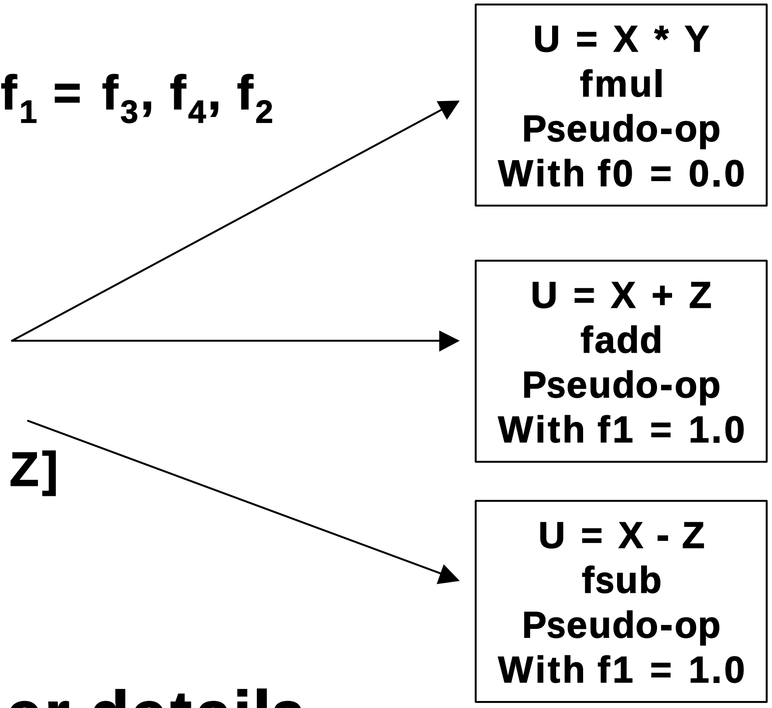
- **Parallel add/subtract**
  - **(qp) padd*n*[.sat] $r_1 = r_2, r_3$**
    - **n = [1,2, or 4]**
    - **Various kinds of saturation**

- **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 * Y + Z]**
  - **Fms [U = X * Y - Z]**
  - **Fnma [U = - (X * Y) + Z]**



U = X * Y
fmul
Pseudo-op
With f0 = 0.0

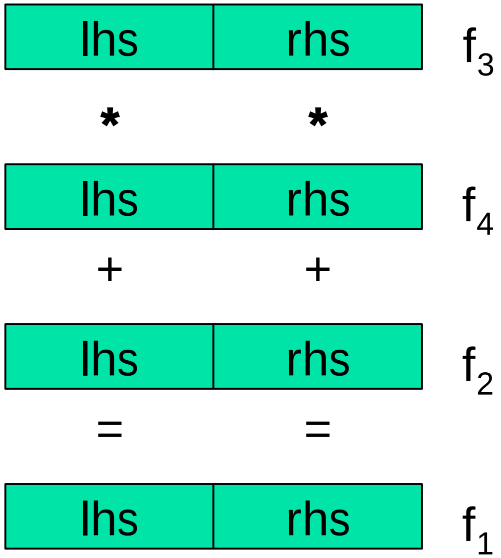
U = X + Z
fadd
Pseudo-op
With f1 = 1.0

U = X - Z
fsub
Pseudo-op
With f1 = 1.0

- **See part 3 for further details**

SIMD Floating-Point

- **Standard instruction:**
  - **(qp) ops.pc.sf** $f_1 = f_3, f_4, f_2$
  - **Valid Operations:**
    - **Fpma [U = X * Y + Z]**
    - **Fpms [U = X * Y - Z]**
    - **Fpnma [U = - (X * Y) + Z]**
  - **See part 3 for further details**



NB: f1 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 $r_1=[r_3], Imm_9$**
  - **(qp) ldf.fsz.fldtype.ldhint $f_1=[r_3], r_2$**
  - **(qp) ldf.fsz.fldtype.ldhint $f_1=[r_3], Imm_9$**

Always post-modify

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

In the case of integer multiply (for instance)

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

Line Prefetch

- Place a cache-line at a given level
  - (qp) lfetch.lftype.lfhint [r_3], r_2
  - (qp) lfetch.lftype.lfhint [r_3], 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** $[r_3] = r_1, Imm_9$
  - **(qp) stf.fsz.fstype.sthint** $[r_3] = f_1$
  - **(qp) stf.fsz.fstype.sthint** $[r_3] = f_1, Imm_9$

No register-based post-modify

- **Valid Sizes:**
  - **Same as Load**

NB: Memory address is the target

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.btype.bwh.ph.dh** **target_{25}/b_2**
  - **(qp) Br.Call. bwh.ph.dh** **b_1= 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)
  - ✓ SIMD Floating Point operations (e.g. fpma)
  - ✓ Memory operations
  - ✓ Move operations
  - ✓ Cache Management operations

How to code instruction operands

- **Two rules:**
  - **Asignment 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: $f_1 = -(f_3 * f_4) + f_2$**
    - **Less Obvious is: Andcm**
      - **AND Complement: r_1 = Input$_2$ & ~Input$_3$**
      - **Complement Input$_2$ or Input$_3$??**

Part 2

Multimedia 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

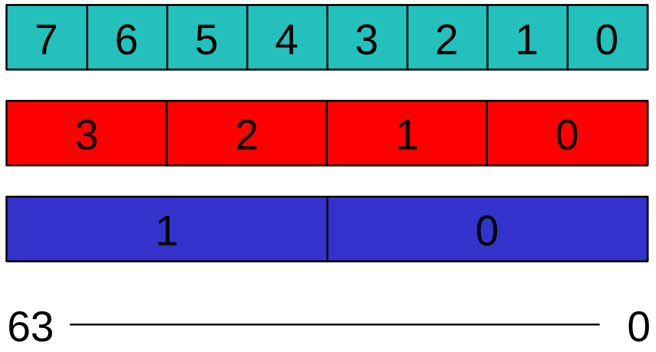
NN Perf. Mon.
Data Reg’s

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 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**

	1B	2B	4B
Padd/Psub	1	2	4
Padd.sus Psub.sus	1	2	-
Pavg[.raz] Pavgsub	1	2	-
Pshladd Pshradd	-	2	-
Pcmp	1	2	4
Pmpy	-	2	-
Pmpyshr	-	2	-
Psad	1	-	-
Pmin/Pmax	1	2	-

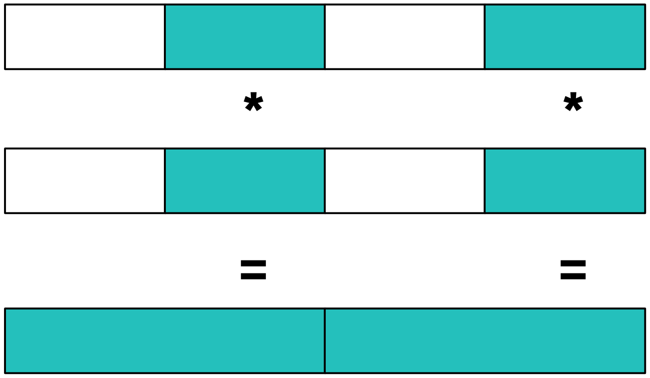
Other instructions

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

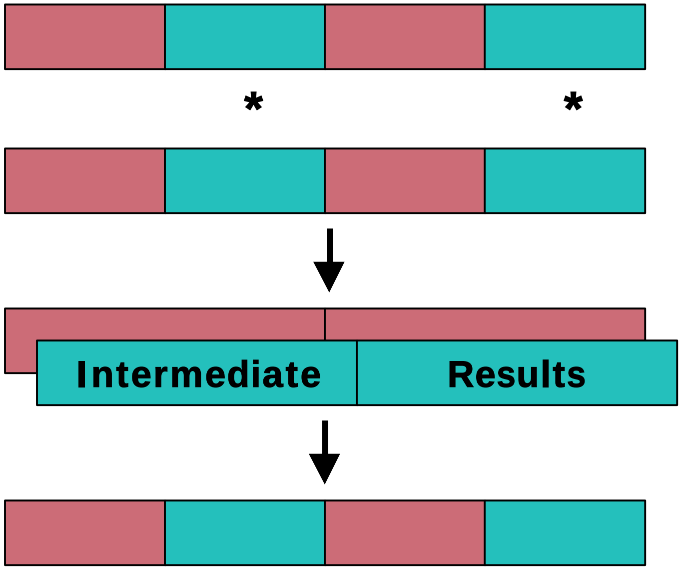
	1B	2B	4B
Pshl/Pshr Pshr.u	-	2	4

	1B	2B	4B
Mix	1	2	4
Mux	1	2	-
Pack.sss	-	2	4
Pack.uss	-	2	-
Unpack	1	2	4

Complex Multimedia - 1



- **Parallel Multiply**
  - **(qp) pmpy2.r $r_1 = r_2, r_3$**
    - **Same instruction for left**

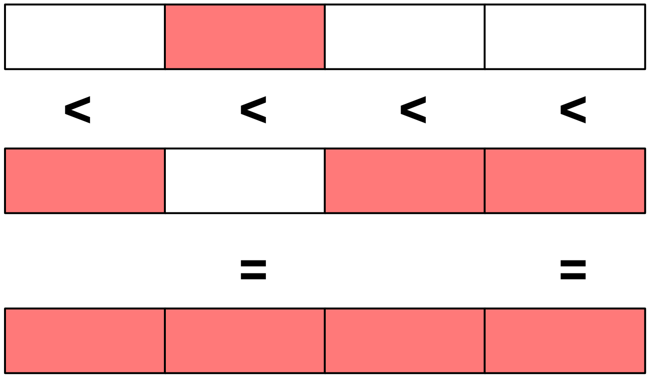


- **Parallel Multiply and Shift Right**
  - **(qp) pmpyshr2[.u] $r_1 = r_2, r_3, count_2$**
    - **Count can be: 0, 7, 15, 16**

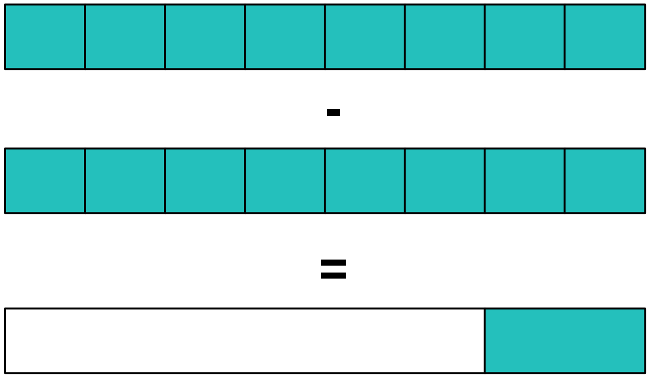
I2 and I1, respectively

Complex Multimedia - 2

- **Parallel Maximum**
  - (qp) pmax2 $r_1 = r_2, r_3$
    - Signed quantities
    - Unsigned if single bytes
      - Pmax1.u



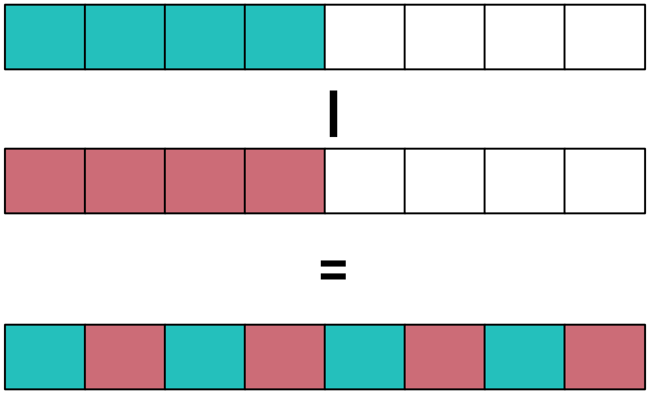
- **Parallel Sum of Absolute Differences**
  - (qp) psad1 $r_1 = r_2, r_3$
    - Absolute difference of each sets of bytes
    - Then sum of these 8 values



Complex Multimedia - 3

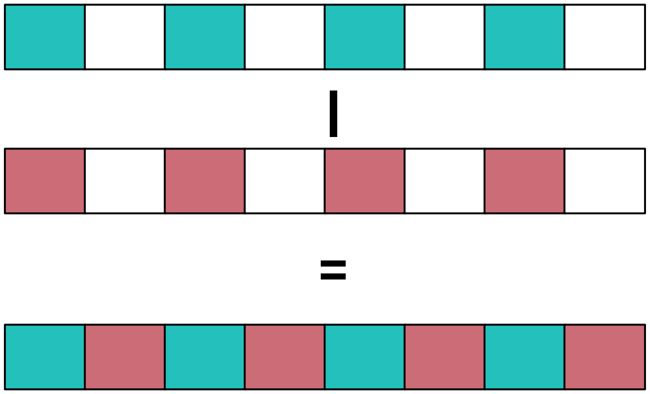
- **Unpack high / low**
  - **(qp) unpack*n*.[h | l] r_1 = r_2, r_3**
    - **“High” uses bits 63-32**
    - **“Low” uses 31-0**
    - **Sizes: 1 / 2 / 4**

Example 1: Unpack1.h



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

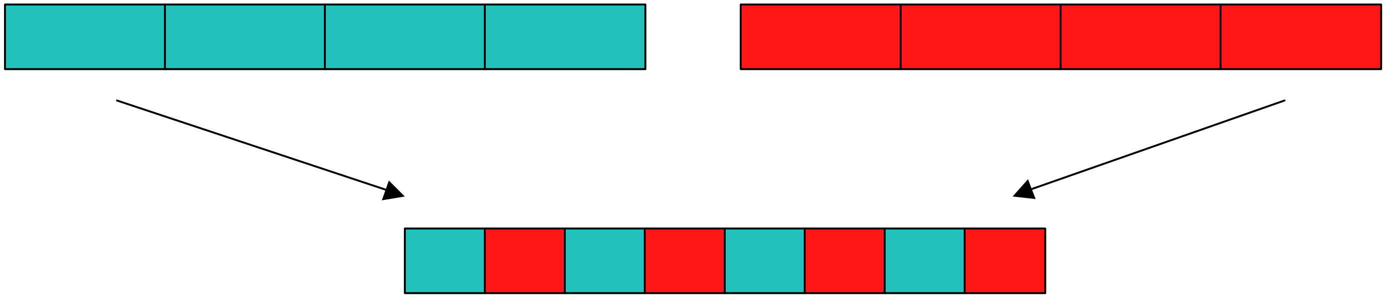
Example 2: Mix1.l



Both are I2

Complex Multimedia - 4

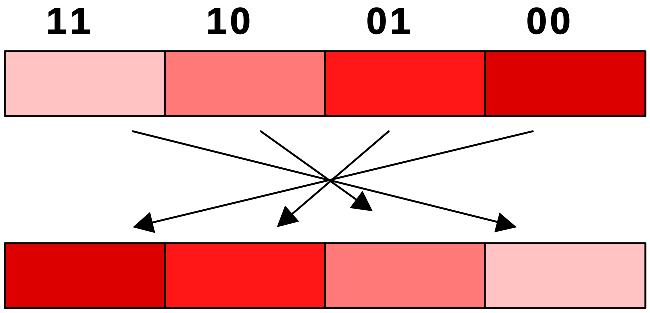
- **Pack w/saturation**
  - **(qp) pack2.sat r_1 = r_2, r_3**
    - **„sat" may be sss/uss**
  - **(qp) pack4.sss r_1 = r_2, r_3**



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)**



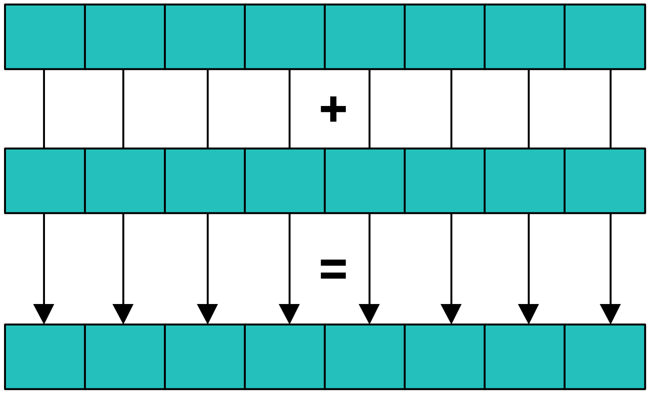
- **Mux1**
  - **Only 'fixed' combinations:**
    - **Reverse (Bytes: 01234567)**
    - **Mix (73516240)**
    - **Shuffle (73625140)**
    - **Alternate (75316420)**
    - **Broadcast (byte 0)**

I4 and I3, respectively

Simple Multimedia - 1

- **Parallel add/subtract**
  - **(qp) padd*n*[.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]**

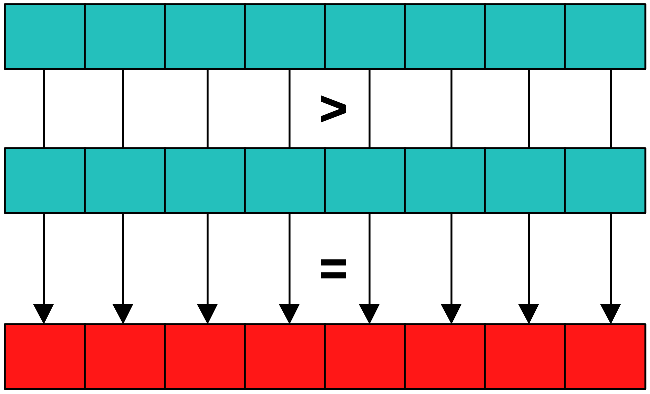
- **Parallel add/subtract**
  - **(qp) padd4 $r_1 = r_2, r_3$**
    - **Modulo arithmetic**



S.Jarp
CERN

Simple Multimedia - 2

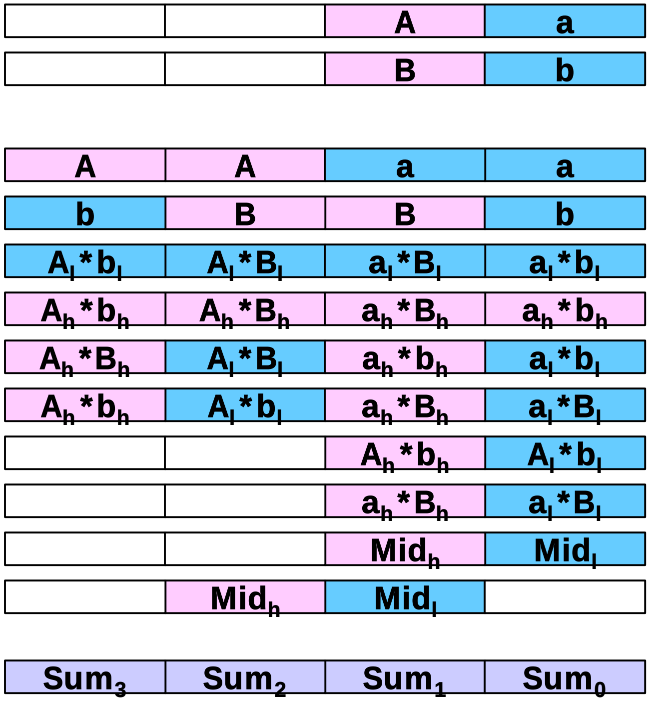
- **Parallel compare**
  - **(qp) pcmp*n*.prel $r_1 = r_2, r_3$**
    - **One/Two/Four byte operands:**
    - **“Prel” may be: eq; gt (signed)**
    - **If true, a mask of 0xff (0xffff or 0xffffffff) is produced**
    - **If false, a mask of zeroes is produced**



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
```



		A	a
		B	b
A	A	a	a
b	B	B	b
A_l*b_l	A_l*B_l	a_l*B_l	a_l*b_l
A_h*b_h	A_h*B_h	a_h*B_h	a_h*b_h
A_h*B_h	A_l*B_l	a_h*b_h	a_l*b_l
A_h*b_h	A_l*b_l	a_h*B_h	a_l*B_l
		A_h*b_h	A_l*b_l
		a_h*B_h	a_l*B_l
		Mid_h	Mid_l
	Mid_h	Mid_l	
Sum_3	Sum_2	Sum_1	Sum_0

Contributed by Walter Misar (TU - Darmstadt)

Multimedia programming

- **MPEG2 motion estimation:**
  - **From IA32 to IA64:**

```
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
```

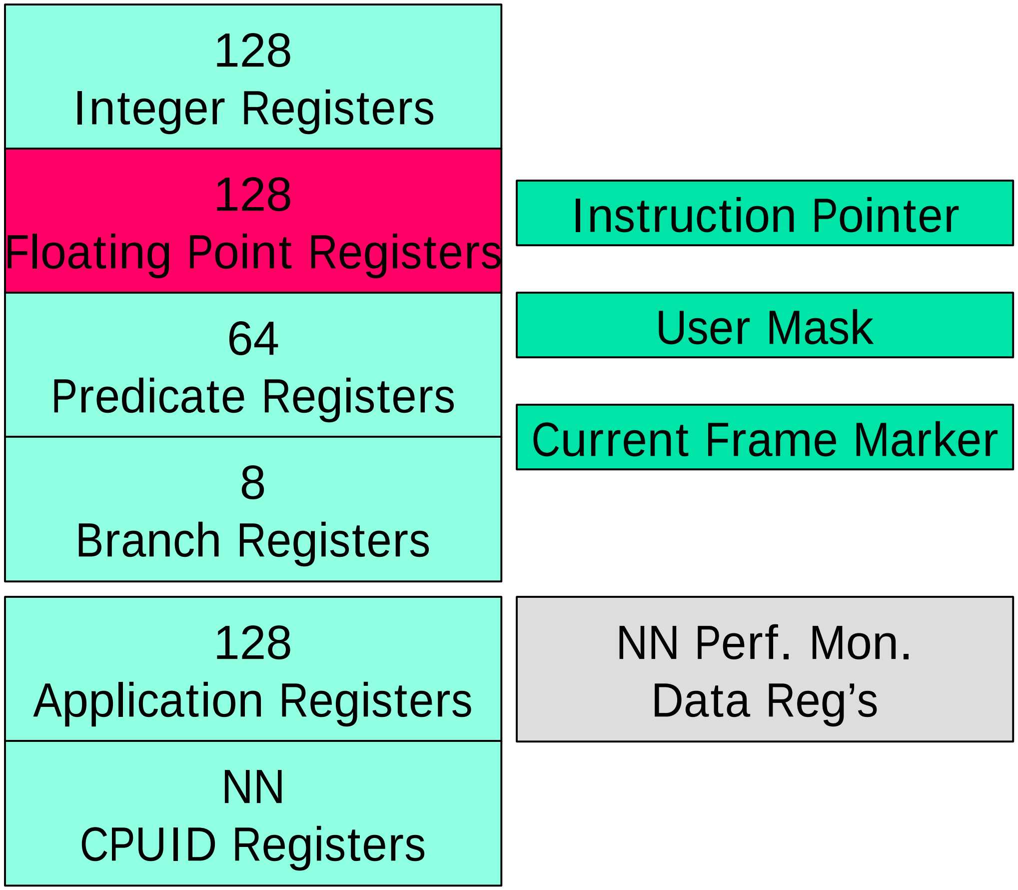
```
Psad_top:  // 16x16 block matching
//Do PSAD for a row, accumulate results
ld8       r32=[r22],r21
ld8       r33=[r23],r21
ld8       r34=[r24],r21
ld8       r35=[r25],r21       ;;
psad1     r32=r32,r34
psad1     r33=r33,r35         ;;
add/padd4  r36=r36,r32
add/padd4  r37=r37,r33
Br.cloop.many.sptk 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)

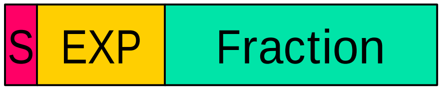
Floating-Point Loads/Stores

- **In matrix form:**

Operand	Ldf.	Ldfp.	Stf.
Single	s	s	s
Double	d	d	d
Integer	8	8	8
Dbl.Ext.	e	-	e
82-bits	fill	-	spill
Post-incr.	Reg/Imm	8/16	Imm

IEEE 754 format

- **Intrinsic construct**
  - **Sign / Unsigned Exponent / Unsigned Significand**
    - **$(-1)^S * 2^E * 1.f$ Example: $-3 = (-1)^1 * 2^1 * 1.5$**
      - **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:	Single Precision:
N bits allow 0 – (2^N-1)	8 bits allow 0 – 255
Bias is defined as: $2^{N-1}-1$	127
Exponent of 0: 0	0
Lowest ‘normal’ exp.: 1	1
Equivalent to $2^{-(2^{N-1}-2)}$	Equivalent to 2^{-126}
Exponent of 1: $2^{N-1}-1$	127
Highest ‘normal’ exp.: 2^N-2	254
Equivalent to $2^{(2^{N-1}-1)}$	Equivalent to 2^{127}
Infinity and NaNs: 2^N-1	255

IA64 number range

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

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

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

- **Register format**
  - **Range of [2^{-65535}, 2^{65536}] corresponds to about [$10^{-19728.0}$, $10^{19728.3}$]**
  - **63-bit accuracy: ~$10^{-19.0}$**

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

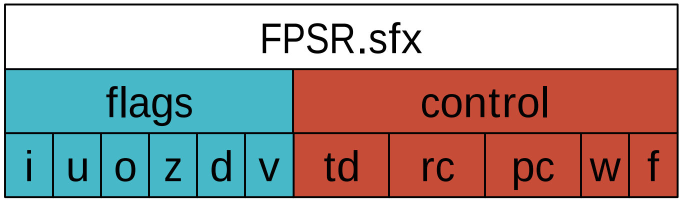
rsv	sf3	sf2	sf1	sf0	traps

FLP Status Register

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

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

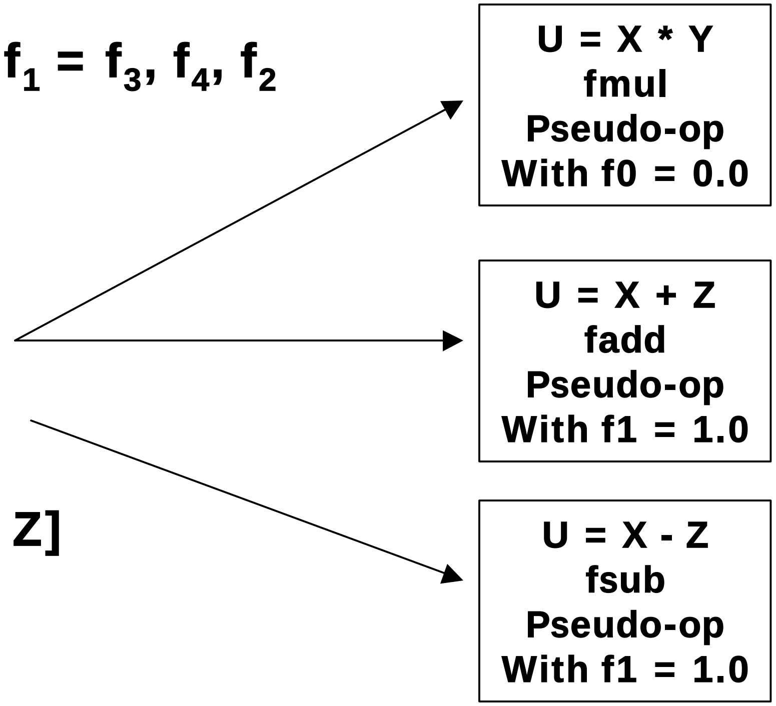
    - **Contains Contro"**
      - **Trap Disabling**
      - **Rounding Control**
      - **Precision Control**
      - **Widest-range-exponent, Flush-to-zero**



FPSR.sfx										
flags						control				
i	u	o	z	d	v	td	rc	pc	w	f

Floating-Point Operations

- Standard instruction:
  - (qp) ops.pc.sf $f_1 = f_3, f_4, f_2$
- Valid Operations:
  - Fma [U = X * Y + Z]
  - Fms [U = X * Y - Z]
  - Fnma [U = - (X * Y) + Z]



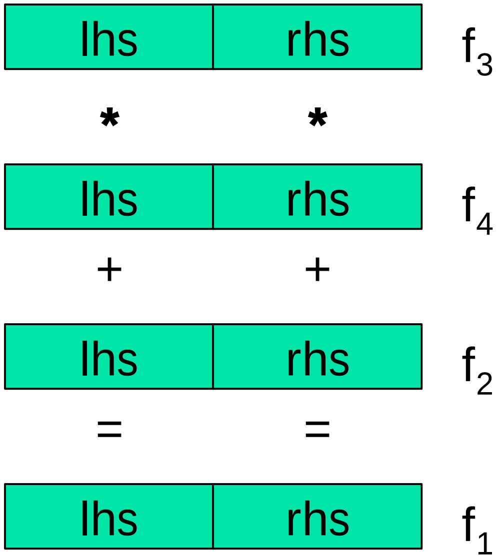
U = X * Y
fmul
Pseudo-op
With f0 = 0.0

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 * Y + Z]**
    - **Fpms [U = X * Y - Z]**
    - **Fpnma [U = - (X * Y) + Z]**



NB: f1 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)merge]
    - Classify [fclass]
- 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

S.Jarp
CERN

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 = double(x * rcp)**
    - **rem = x – z*y   // remainder**
    - **z = double(z_0 + rem*rcp)**

  - **Cost:**
    - **10 operations (8 groups)**

Accurate for Double Precision Results

FLP Divide

- **Actual code:**

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

```
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
```

- **Issue:**
  - Long latency

What if we need just the remainder ?

Macro as already shown

Integer remainder

- **Steps needed:**
  - **Transfer variables**
  - **Convert to FLP**
  - **Do the Division**
  - **Compute remainder**
  - **Convert to integer**
  - **Transfer back**
- **Issue:**
  - **Even longer latency**

```
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
;;
          fcvt.fx.trunc.s1 f6=f6 // convert to integer
;;
          getf.sig  r6=f6    // remainder
```

Macro as already shown

S.Jarp
CERN

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

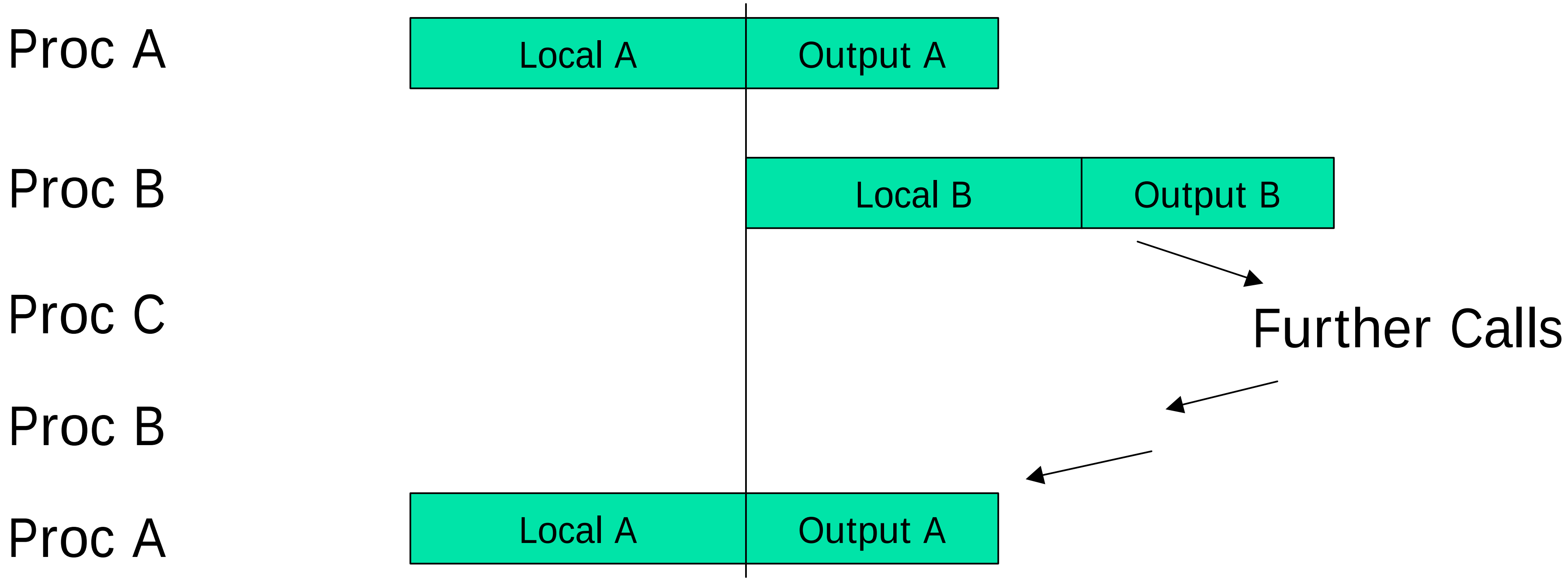
- Exploit (in priority order)
  - Architectural support
    - Modulo Scheduling support
      - Predication
      - Register Rotation (Large Register Files)
    - Full access to other features
      - SIMD, Prefetching, Load pair instructions, etc.
  - Micro-architecture
    - Number of parallel slots; Execution units; Latencies
    - Cache sizes, Bandwidth
  - Tricks
    - For increased speed
      - integer multiplication via shladd-sequences, etc.
    - For balanced execution capability (FLP INT)

“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**

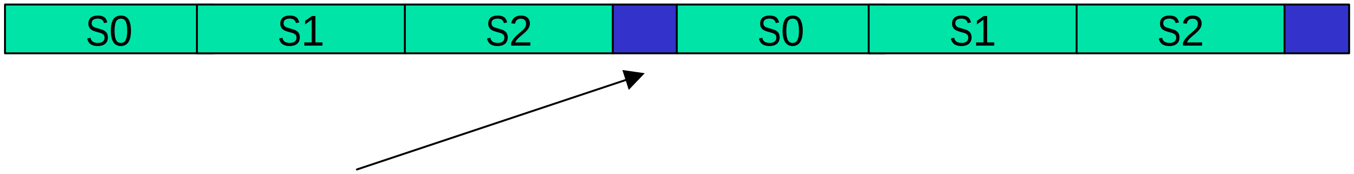
Register Stack

- **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)**



Execution Width

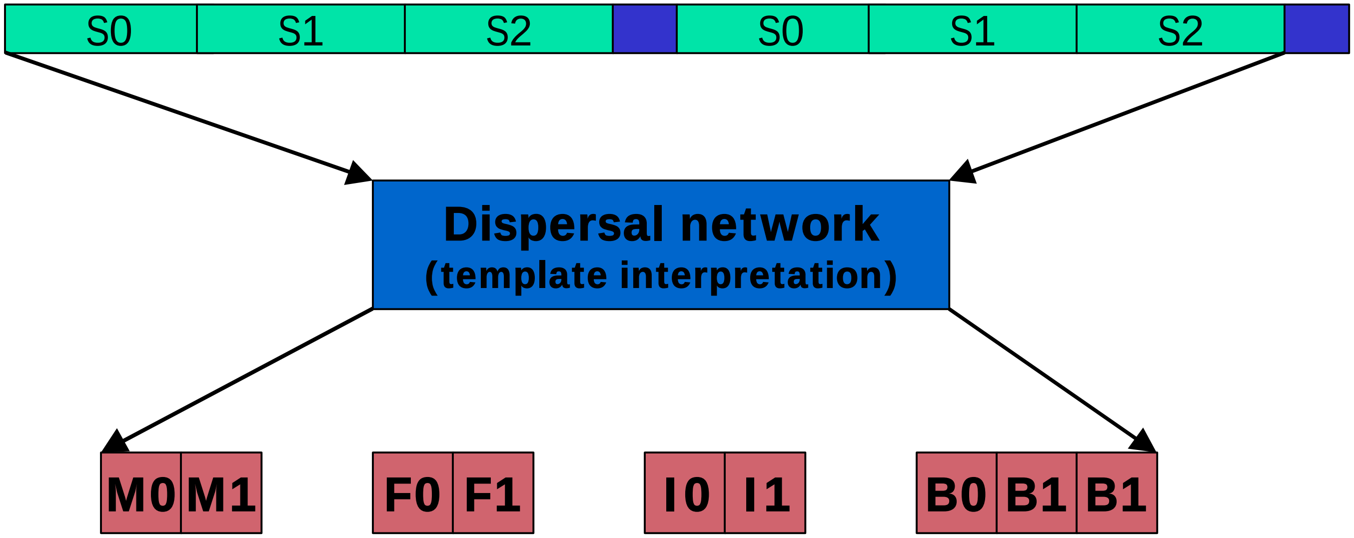
- A given implementation could be N wide
  - Itanium/Merced is implemented as a “two-banger”
    - 6 parallel instructions
      - Major enhancement compared to IA-32
    - But,
      - If nothing useful is put into the syllables, they get filled as NOPs



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

Instruction Delivery

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



<u>9 available ports in total</u>

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 + ....
S = R * ...
X = S - ...
Y = X/...
Z = Y +
```

Initial Example

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

3 groups
in
3 cycles

```
getval:
                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
```

MLX

M+MI

MIB

Explicit
Stop bit
Or
Enforced
Bundle
Break

Parallel Compares

- **Instruction format:**
  - (qp) cmp.crel.ctype $p_1, p_2 = r_2, r_3$
  - (qp) cmp.crel.ctype $p_1, p_2 = Imm_8, r_3$
  - (qp) cmp.crel.ctype $p_1, p_2 = r0, r_3$

- **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

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     ;;
```

4 cycles

  - **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     ;;
```

1+ cycle

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

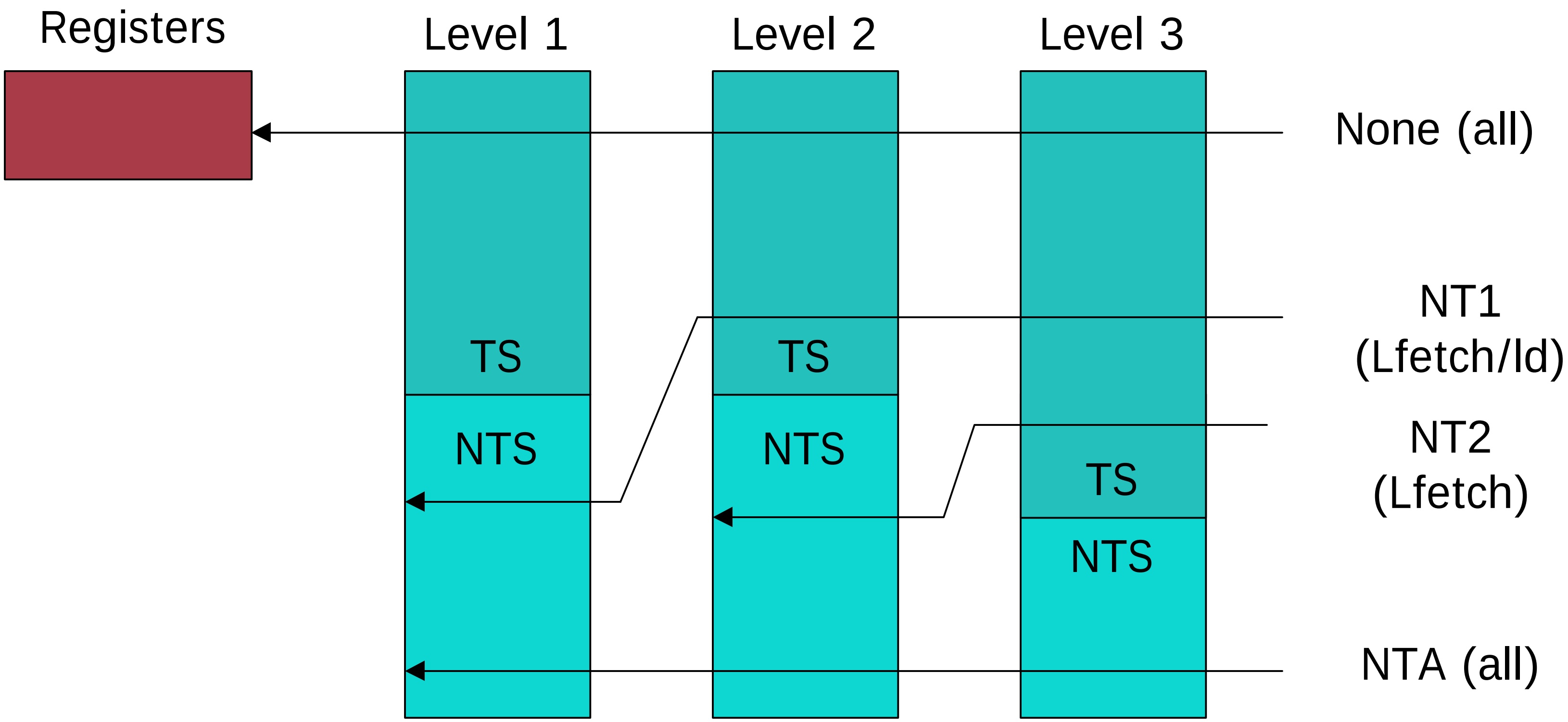
- **Place a cache-line at a given level**
  - **(qp) lfetch.lftype.lfhint [r_3], r_2**
  - **(qp) lfetch.lftype.lfhint [r_3], Imm_9**

- **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**



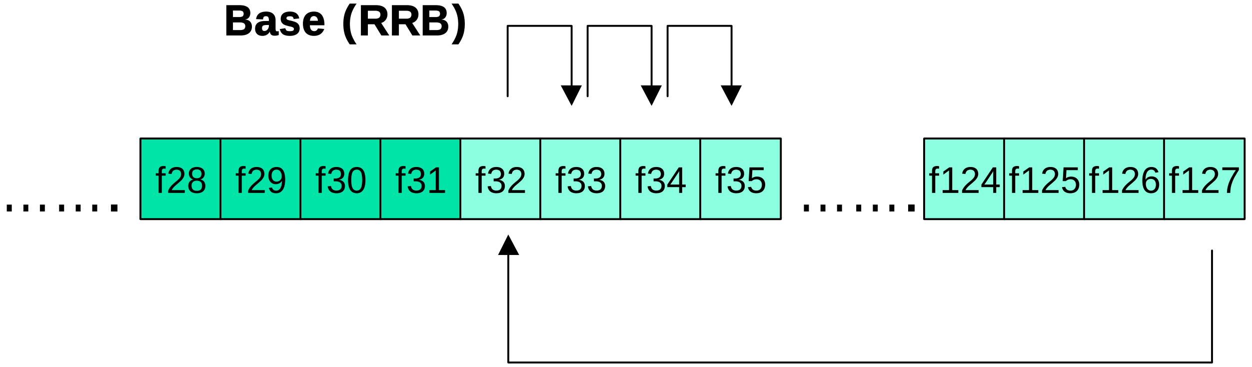
Modulo Scheduled Loop

- 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

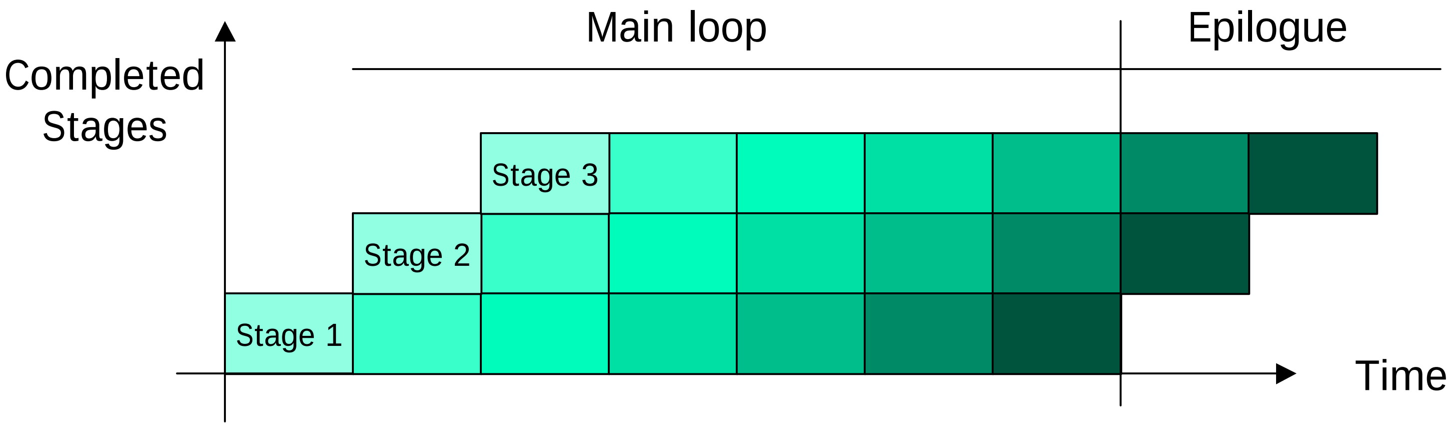
- Upper 75% rotate (when activated):
  - General registers (r32-r127)
  - Floating Point Registers(f32-f127)
  - Predicate Registers (p16-p63)

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



Modulo Loop - 2

- **Graphical representation**
  - **7 loop traversa" desired**
  - **Skewed execution**
    - **Stage 2 relative to Stage 1**
    - **Stage 3 relative to Stage 2**



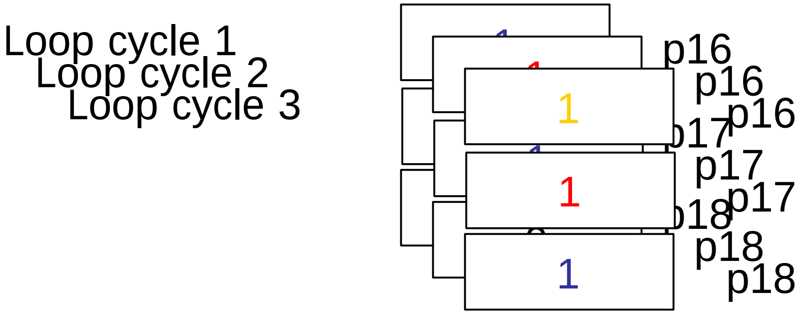
Modulo Loop - 3

- 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

S.Jarp
CERN

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**



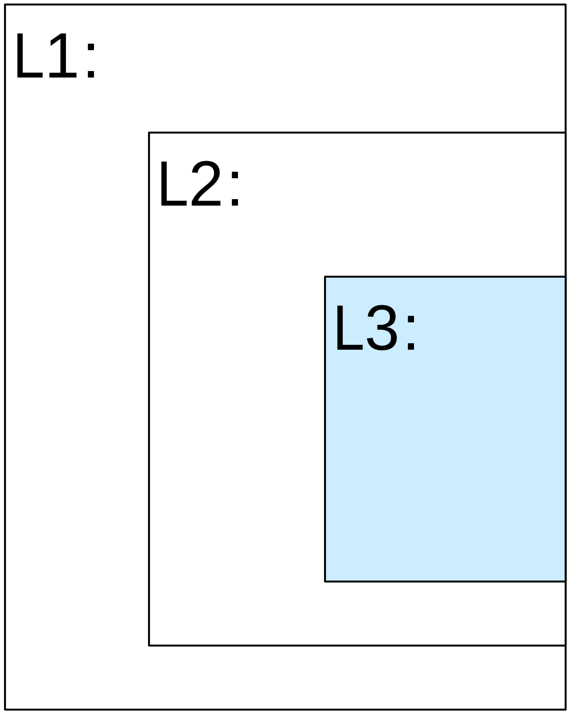
Modulo Loop - 5

- **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**

```
        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

Type	Instructions	Category
A1	Add; Sub (Register) And; Andcm; Or; Xor	Integer ALU
A2	Shladd	"
A3	Sub (Immediate) And; Andcm; Or; Xor	"
A4	Adds	"
A5	Addl	"
A6	Compare (Reg.)	Int. Compare
A7	Compare to Zero	"
A8	Compare (Imm.)	"
A9	Padd; Psub; Pavg; Pcmp	Multimedia
A10	Pshladd; Pshradd	"

S.Jarp
CERN

Appendix 1b

- I-instructions
  - Part 1
    - Multimedia and Variable Shifts
    - Integer Shifts

Type	Instructions	Category
I1	Pmpyshr	Multimedia
I2	Pmpy; Mix; Pack; Unpack Pmin; Pmax; Psad	"
I3	Mux1	"
I4	Mux2	"
I5	Shr; Pshr (Variable)	"
I6	Pshr (Fixed)	"
I7	Shl; Pshl (Variable)	"
I8	Pshl (Fixed)	"
I9	Population Count	"
I10	Shrp	Int. Shift
I11	Extract	"
I12	Zero and deposit	"
I13	Zero and deposit (Imm.)	"
I14	Deposit (Imm.)	"
I15	Deposit	"

Appendix 1c

- I-instructions
  - Part 2
    - Miscellaneous

Type	Instructions	Category
I16	Test Bit	Test Bit
I17	Test Nat	"
I18	Move Long	Int. Misc.
I19	Break.i; Nop.i	"
I20	Chk.s.i	"
I21	Move to BR	Int. Move
I22	Move from BR	"
I23	Move to Predicate (Reg.)	"
I24	Move to Predicate (Imm.)	"
I25	Move from PR/IP	"
I26	Move to AR (Reg.)	"
I27	Move to AR (Imm.)	"
I28	Move from AR	"
I29	Sign/Zero Extend; Compute Zero Index	Int. Misc.

Appendix 1d

- **M-instructions**
  - **Load**
  - **Store**
  - **Prefetch**

Type	Instructions	Category
M1	Integer Load	Load / Store
M2	Integer Load (PI via reg.)	"
M3	Integer Load (PI via imm.)	"
M4	Integer Store	"
M5	Integer Store (PI via imm.)	"
M6	Floating-Point Load	"
M7	FLP Load (PI via reg.)	"
M8	FLP Load (PI via imm.)	"
M9	FLP Store	"
M10	FLP Store (PI via imm.)	"
M11	FLP Load Pair	"
M12	FLP Load Pair (PI via imm.)	"
M13	Line prefetch	Prefetch
M14	Line prefetch (PI via reg.)	"
M15	Line prefetch (PI via imm.)	"

Appendix 1e

- M-instructions
  - Miscellaneous

Type	Instructions	Category
M16	(Cmp and) Exchange	Semaphore
M17	Fetch and Add	"
M18	Setf	Set/Get
M19	Getf	"
M20	Chk.s.m (INT)	Speculation
M21	Chk.s (FLP)	"
M22	Chk.a.nc/clr (INT)	"
M23	Chk.a.nc/clr (FLP)	"
M24	Sync; Fence; Serialize	Synchr.
M25	Flushrs	"
M26	Invala.e (INT)	"
M27	Invala.e (FLP)	"
M28	Flush cache	"

Appendix 1f

- **M-instructions**
  - **Register moves**
  - **Misc.**

Type	Instructions	Category
M29	Move to AR (Reg.)	Mem.Mov.
M30	Move to AR (Imm.)	"
M31	Move from AR	"
M32		
M33		
M34	Alloc	M.Misc.
M35	Move to PSR	"
M36	Move from PSR	"
M37	Break.m; Nop.m	"
M38		
M39		
M40		
M41		
M42		
M43	Move from Indirect Reg.	Mem.Mgm.
M44	Set/Reset User Mask	"

Appendix 1g

- B-instructions
  - Whole set

Type	Instructions	Category
B1	IP-relative branch	Branch
B2	IP-rel. Counted Branch	"
B3	IP-rel. Call	"
B4	Indirect Branch (B-reg.)	"
B5	Indirect Call (B-reg.)	"
B6		
B7		
B8	Clrrrb	Br.Misc.
B9	Break.b / Nop.b	Br.Nop.

S.Jarp
CERN

Appendix 1h

- F-instructions
  - Whole Set
    - Arithmetic
    - Compare and Classify
    - Approximations
    - Miscellaneous
    - Convert
    - Status Fields

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

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**