CSE 661 – Parallel and Vector Architectures

Research Project

Problem Statement

The main idea of this research project is to enable a parallel loop to execute across all cores, taking advantage of the multiplicity of resources provided in a multicore processor. With the introduction of few instructions and minimal extra hardware support, a thread running in a single core will be able to broadcast a parallel loop to all cores. This mode of execution will be supported completely in the microarchitecture. The operating system "sees" the parallel loop as one thread, rather than multiple threads running on multiple cores.

A chip multiprocessor with *N* cores and a shared L2-cache is shown below. Each core is capable of executing a thread scheduled by the operating system, referred here as a *root* thread, plus few additional threads created by the hardware to speedup the execution of parallel loops. An L2 cache is shared by all the cores. To optimize the bandwidth and latency, the L2 cache is divided into *M* independent banks that operate in parallel. The *N* cores communicate with the *M* cache banks using the on-chip interconnect. There is nothing new, except the ability to execute parallel loop instructions across all cores. A parallel loop instruction is *effectively converted into N scalar instructions* by distributing its work on the *N* CPU cores.

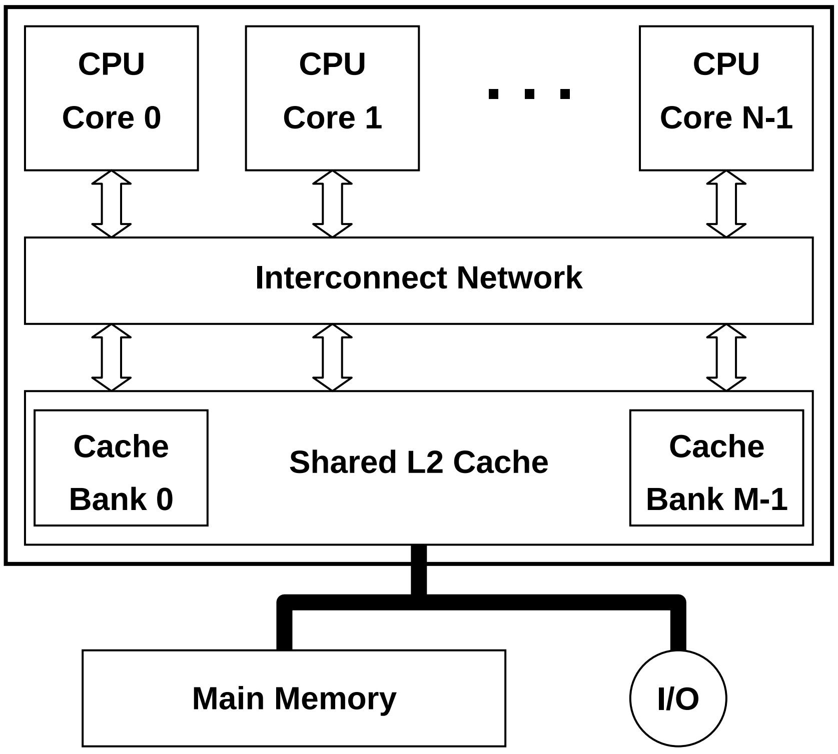


Figure 1: Chip multiprocessor with *N* cores and a shared L2 cache with *M* banks

Parallel Counter-Controlled Loops

Consider the execution of the following loop where *x* and *y* are vectors residing in memory and *a* is a scalar value. This is the DAXPY loop that forms the inner loop of the Linpack benchmark for performing Gaussian elimination:

```
for (i=0; i<n; i++) y[i] = a * x[i] + y[i];
```

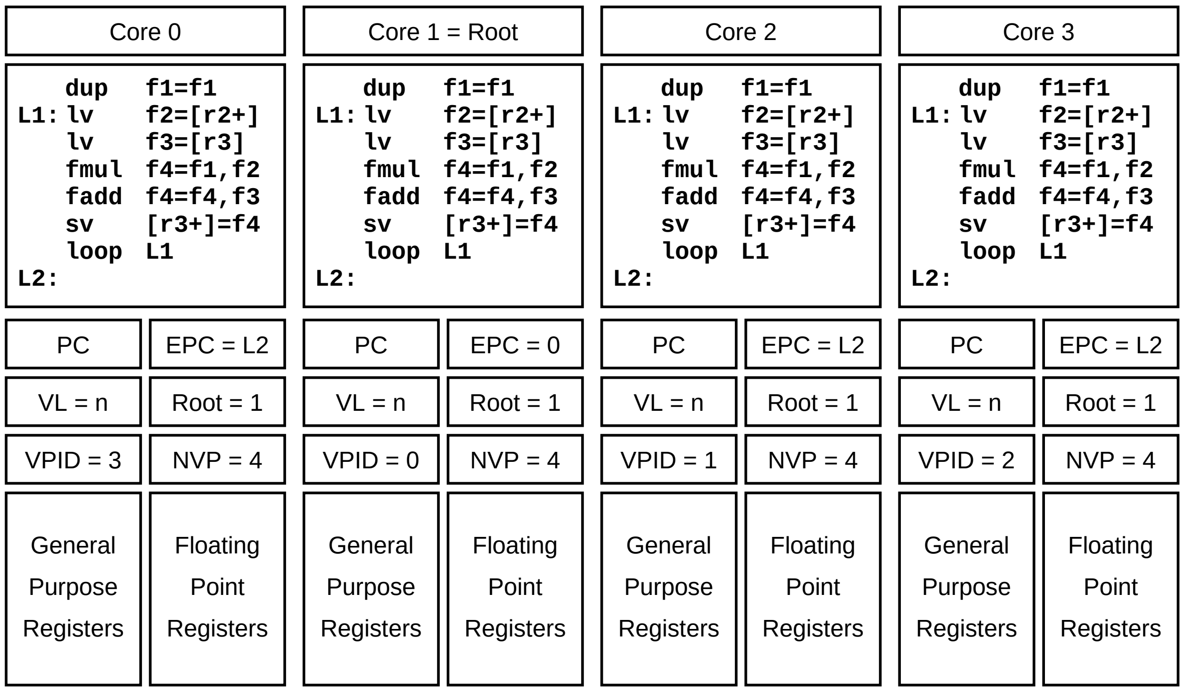
The iterations of the above loop can be executed in parallel. Assume that the base addresses of arrays *x* and *y* are in registers **r2** and **r3** respectively, and the scalar values *n* and *a* are loaded in registers **r1** and **f1** respectively. Then, the above loop can be translated as follows:

```
   vp      r0 = r1, L2     ; allocate virtual processors
   dup     f1 = f1         ; duplicate f1 = a in all VPs
L1:
   lv      f2 = [r2+]      ; load vector x into f2 in all VPs
   lv      f3 = [r3]       ; load vector y into f3 in all VPs
   mul     f4 = f1, f2     ; multiply: a * x[i] in all VPs
   add     f4 = f4, f3     ; add: a * x[i] + y[i] in all VPs
   sv      [r3+] = f4      ; store vector f4 at address y
   loop    L1
L2:
```

The VP Instruction

The above loop appears to be sequential, but is in fact a parallel loop. The **vp** (Virtual Processor) instruction allocates virtual processors to execute a parallel loop. Each virtual processor is a hardware thread that includes integer and floating-point register files, a program counter, and some additional control registers. The **vp** instruction specifies the vector length and the label address at which to terminate parallel execution. It returns the number of allocated virtual processors (NVP). The **vp** instruction enables the exploitation of data-level parallelism across multiple cores.

In addition to allocating hardware contexts across all cores, the **vp** instruction initializes the vector length (VL) register in all hardware contexts with the specified number of iterations, the virtual processor identification (VPID) register with a unique number, the virtual processor count (NVP) register with the count of VPs (a power of 2), and the Root register with the root core number. The **vp** instruction also initializes the end program counter (EPC) register in all the allocated VP contexts with the label address that marks the end of the parallel loop, and the program counter (PC) registers with the address of next instruction to launch parallel loop execution. This is illustrated in Figure 2, where Core 1 is the root core that issued the execution of the **vp** instruction. The root core always have VPID = 0. The instructions appearing after **vp** will be executed as asynchronous parallel threads (not in lockstep) on all the virtual processors until the end label is reached. This mode of execution is more flexible than the lockstep vector execution mode implemented in vector processors.



Core 0

```
    dup   f1=f1
L1: lv    f2=[r2+]
    lv    f3=[r3]
    fmul  f4=f1,f2
    fadd  f4=f4,f3
    sv    [r3+]=f4
    loop  L1
L2:
```

Core 1 = Root

```
    dup   f1=f1
L1: lv    f2=[r2+]
    lv    f3=[r3]
    fmul  f4=f1,f2
    fadd  f4=f4,f3
    sv    [r3+]=f4
    loop  L1
L2:
```

Core 2

```
    dup   f1=f1
L1: lv    f2=[r2+]
    lv    f3=[r3]
    fmul  f4=f1,f2
    fadd  f4=f4,f3
    sv    [r3+]=f4
    loop  L1
L2:
```

Core 3

```
    dup   f1=f1
L1: lv    f2=[r2+]
    lv    f3=[r3]
    fmul  f4=f1,f2
    fadd  f4=f4,f3
    sv    [r3+]=f4
    loop  L1
L2:
```

Figure 2: Executing a parallel loop on four cores

When the program counter (PC) reaches the end label program counter (EPC), the virtual processor terminates execution and the hardware context is freed. Eventually all virtual processors will free their hardware context, except for the root thread, which continues normal execution after the end of the parallel loop. A special case occurs when the vector length is equal to 0. In this case, no virtual processor is allocated and the **vp** instruction simply becomes a jump to the end label, skipping all instructions in a parallel loop.

If all hardware contexts are allocated and the **vp** instruction fails to allocate new ones, then the parallel loop will be executed sequentially in the root core, rather than as parallel threads in virtual processors.

Support for Packed Data Types

64-bit registers can pack multiple data of smaller sizes. This packing optimizes the use of registers and adds sub-register data parallelism. Four integer register formats, specified as opcode extensions, are defined: *long word* (**`.l`** extension), *packed words* (**`.w`** extension), *packed half words* (**`.h`** extension), and *packed bytes* (**`.b`** extension).

64-bit Long word							
32-bit Word 1				32-bit Word 0			
Half word 3		Half word 2		Half word 1		Half word 0	
Byte7	Byte6	Byte5	Byte4	Byte3	Byte2	Byte1	Byte0

Four integer register formats are given to the same general-purpose register

There is *no distinction between scalar and vector instructions*. The same integer arithmetic instruction can operate on all packed formats, in addition to the long word format. Consider the **add** instruction. Four executions can result depending on the register format, as shown below. The same 64-bit datapath can be internally partitioned inside the ALU to produce different results, depending on the register format. If the register format is not specified, it defaults to long word (**`.l`** extension).

r1	64-bit Long word
	+
r2	64-bit Long word
	=
r3	64-bit Long word

`add.l r3 = r1, r2 # long words`

r1	32-bit Word 1	32-bit Word 0
	+	+
r2	32-bit Word 1	32-bit Word 0
	=	=
r3	32-bit Word 1	32-bit Word 0

`add.w r3 = r1, r2 # packed words`

r1	Halfword 3	Halfword 2	Halfword 1	Halfword 0
	+	+	+	+
r2	Halfword 3	Halfword 2	Halfword 1	Halfword 0
	=	=	=	=
r3	Halfword 3	Halfword 2	Halfword 1	Halfword 0

`add.h r3 = r1, r2 # half words`

r1	Byte7	Byte6	Byte5	Byte4	Byte3	Byte2	Byte1	Byte0
	+	+	+	+	+	+	+	+
r2	Byte7	Byte6	Byte5	Byte4	Byte3	Byte2	Byte1	Byte0
	=	=	=	=	=	=	=	=
r3	Byte7	Byte6	Byte5	Byte4	Byte3	Byte2	Byte1	Byte0

`add.b r3 = r1, r2 # packed bytes`

Four executions of the **add** instruction based on the register format

For floating-point instructions, two register formats are defined as opcode extensions: *double-precision* (**`.d`** extension) and *packed single-precision* (**`.s`** extension). Consider the floating-point **add** instruction. Two executions can result depending on the register format.

f0	64-bit double-precision float
	+
f1	64-bit double-precision float
	=
f2	64-bit double-precision float

`add.d f2 = f0, f1 # double float`

f0	Single-precision float 1	Single-precision float 0
	+	+
f1	Single-precision float 1	Single-precision float 0
	=	=
f2	Single-precision float 1	Single-precision float 0

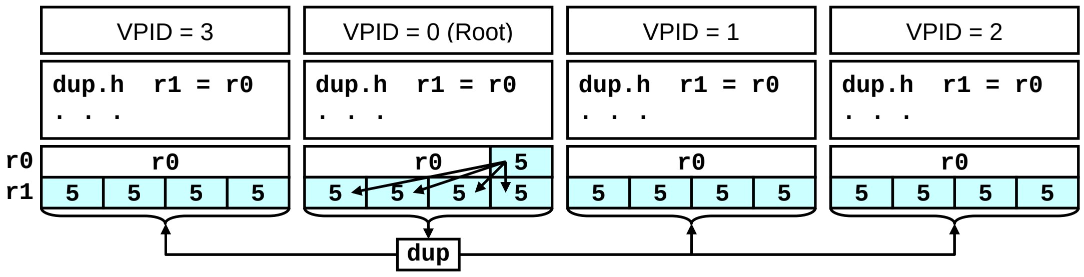
`add.s f2 = f0, f1 # packed floats`

Two executions of the floating-point add instruction based on the floating-point register format

The DUP Instruction

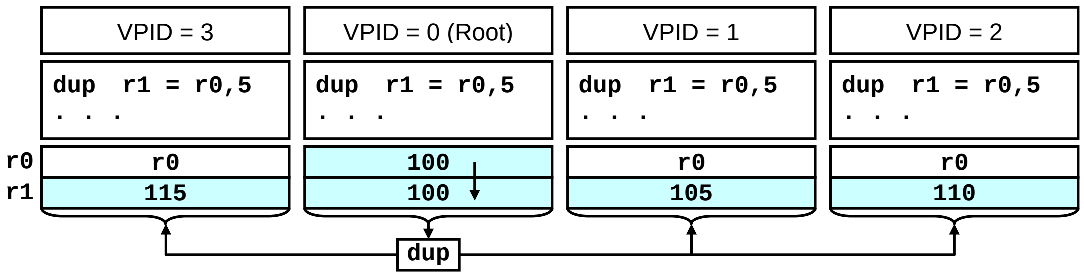
In its simplest form, the **`dup`** (duplicate) instruction broadcasts a register from a root thread to and all the allocate VP threads. The **`dup`** instruction accomplishes register-to-register communication. It acts as a non-blocking send and injects a token into the network when issued by a root thread. It acts as a blocking-receive in all the allocated VP threads.

The **`dup`** instruction is also used to replicate a narrow data element and pack it in a destination register at the root thread, and then broadcast the destination register to the allocated VP threads. The **`.f`** extension is used to specify the narrow data type: **`dup.b`** (duplicate byte), **`.h`** (duplicate half word), **`.w`** (duplicate word), or **`.l`** (duplicate long word). If the **`.f`** extension is not specified, it is assumed to be **`.l`**, which duplicates a general-purpose register. For floating-point registers, the **`.f`** extension can be **`.s`** (duplicate single-precision float), or **`.d`** (duplicate double-precision float). If omitted, it is assumed to be **`.d`**. Duplicating a narrow data element is useful even if no VP thread is allocated. The **`dup`** instruction becomes a simple register-to-register move instruction within the root VP.



Duplicating the least-significant half-word of r0 into r1 at root and broadcasting r1 to all VP threads

Another use of the **`dup`** instruction is to broadcast a different computed value to each VP thread. A second source operand is used to compute a series of values. For example, if **`r0`** is **`100`** then **`dup r1 = r0,5`** produces the following series: **`100`**, **`105`**, **`110`**, **`115`** across all VPs. The value **`100`** is broadcast to all VPs and computed as **100 + VPID × 5** at each VP.



Broadcasting and computing a series of values across all VPs

A more general example is: **`dup.h r1 = r0, 5`**. First, **`r1`** is computed from **`r0=100`** at the root VP as **`r1=[115,110,105,100]`**. Four half words are packed into **`r1`** because the **`.h`** extension is used. Then, **`r1`** at the root is broadcast to all VPs and added to **`VPID×5<<2`**. The constant 5 is shifted left 2 bits (multiplied by 4) because the **`.h`** extension is used. This produces the following computed values of **`r1`**: **`[135,130,125,120](VPID=1)`**, **`[155,150,145,140](VPID=2)`**, and **`[175,170,165,160](VPID=3)`**.

The syntax of the **`dup`** instruction is: **`dup.f rd = rs, rt|im`**

The computed values are: **`rd = rs + VPID × (rt|im) << f`**

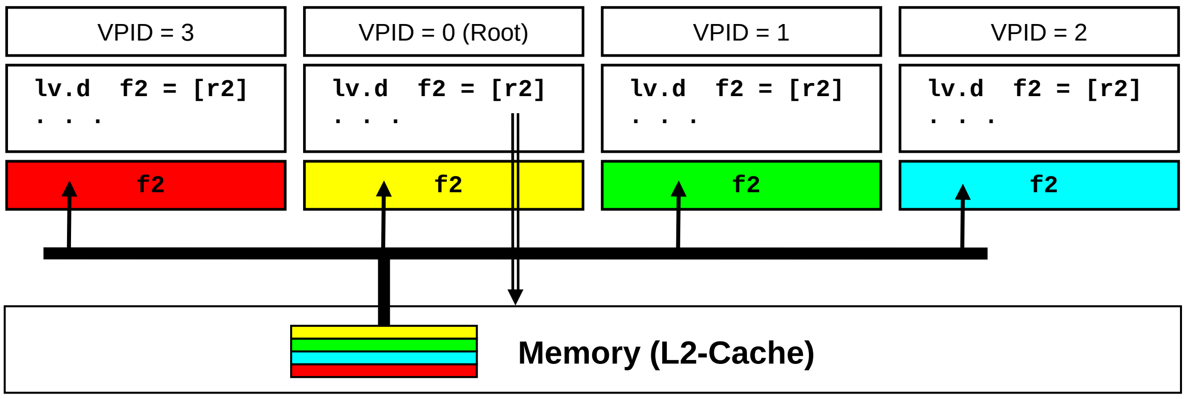
Vector Load and Store

The load and store vector instructions are defined as follows:

```
lv.f      rd = [rs]      # load vector
sv.f      [rs] = rt      # store vector
```

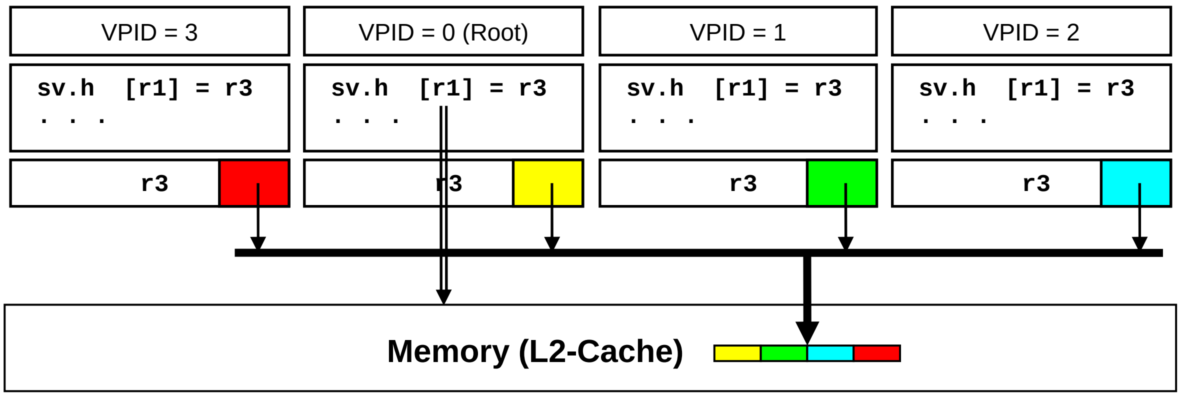
The **`.f`** extension can be **`.b`** (byte), **`.h`** (half word), **`.w`** (word), or **`.l`** (long word). For floating-point, the **`.f`** extension can be **`.s`** (single-precision) or **`.d`** (double-precision). Destination register **`rd`** is replaced by **`fd`**, and source register **`rt`** is replaced by **`ft`**.

The **`lv`** (load vector) instruction loads a contiguous block of memory and distributes the vector elements onto the virtual processor registers. Register **`[rs]`** specifies the vector address. Each VP receives one element of the vector. The **`lv`** instruction is issued only by the root VP. The other VPs act as receivers as shown below. The elements are received in the order specified by VPID. The root VP (with VPID = 0) receives the first element.



The **`sv`** (store vector) instruction gathers the vector elements and stores them in a contiguous block of memory at the specified vector address. Register **`[rs]`** at the root specifies the vector address. The elements of the vector are gathered in the order specified by VPID. They can be gathered at the root core or in the L2-Cache depending on implementation.

It is also possible to load/store vectors of narrow elements such as bytes, half words, and words. The following example shows how the **`sv`** instruction can store a vector of half words, whose elements are distributed onto the virtual processor registers. The **`.h`** extension is used to store the least-significant half word of each register in memory.



The **`lv`** and **`sv`** instruction make use of the VL (Vector Length) and NVP (Number of Virtual Processors) registers. If VL >= NVP then NVP elements are loaded/stored. If VL < NVP then VL elements are loaded/stored.

Addressing Modes

The **`lv`** and **`sv`** instructions use the *register-indirect* addressing mode. Register **`[rs]`** at the root core specifies the memory address. No immediate can be specified as a displacement.

Two update addressing modes are also defined:

`[rs+]` notation indicates that the address should be updated as: **`rs = rs + NVP<<f`**.

`[rs-]` notation indicates that the address should be updated as: **`rs = rs - NVP<<f`**.

Where **`NVP`** is the number of allocated VPs and **`f`** is a scale factor specified by the **`.f`** extension: **`f=0`** for **`.b`**, **`1`** for **`.h`**, **`2`** for **`.w`** or **`.s`**, and **`3`** for **`.l`** or **`.d`**.

The last addressing mode can be used to load or store vectors in reverse order.

Vectors of Different Element Sizes

There are situations in which vectors processed in a given loop have different element sizes. For example, suppose we want to add A = A + B, where A is a vector of half words and B is a vector of bytes. We can load, compute, and store the vectors as shown below, where **`r1`** contains the address of array A and **`r2`** contains the address of array B.

```
lv.h    r3 = [r1]    # load vector A: 1 half word per VP
lv.b    r4 = [r2]    # load vector B: 1 byte per VP
add     r5 = r3, r4  # do the addition
sv.h    [r3] = r5    # store 1 half word from each VP
```

The above code will load one half-word element of vector A into register **`r3`** in each virtual processor. The number of elements loaded simultaneously and processed in parallel is equal to the number of virtual processors. A half-word element occupies the lower 16 bits of a register. It is sign-extended to fill the entire register. Similarly, one byte element of vector B is loaded into register **`r4`** in each virtual processor and is also sign-extended.

We can do better by loading multiple packed vector elements into each register. We use **`lv.l`** to load a long word which packs four half words of vector A into register **`r3`** in each virtual processor. Similarly, we use **`lv.w`** to load a word which packs four bytes of vector B into register **`r4`** in each virtual processor. The four packed bytes, which occupy the lower half of **`r4`**, can be unpacked to become four packed half words. The **`unpk.h`** instruction is used for this purpose. The **`add.h`** instruction is used to carry the addition in parallel on four packed half words in register **`r3`** and **`r4`**.

```
lv.l    r3 = [r1]    # load vector A: 4 half words per register
lv.w    r4 = [r2]    # load vector B: 4 bytes per register
unpk.h  r4 = r4      # Unpack 4 bytes into 4 signed half words
add.h   r5 = r3, r4  # add four packed bytes in parallel
sv.l    [r3] = r5    # store 4 half words from each register
```

The speedup of a parallel loop comes from two factors: the packing done at the level of registers and the number of cores (or virtual processors) involved in the parallel execution.

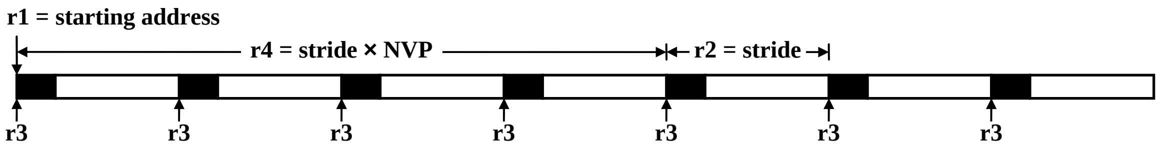
Vectors with a Non-Unit Stride

The **lv** and **sv** instructions can load and store only contiguous vectors. The implementation of these two instructions can always be optimized at the hardware level. Pre-fetching techniques can reduce the load delay. The vector length is a hint to stream the load from memory for long vectors. The cache footprint can be minimized or eliminated if the vector is loaded once, or stored with no future use.

To handle vectors with a non-unit stride, one can define special load/store instructions that can transfer vectors with a non stride. However, such instructions will add more complexity to the hardware. Instead, scalar load/store instructions can be executed in parallel in all VPs.

For example, consider taking the sum of elements in a vector with a non-unit stride. If the starting address of the column is loaded into register **r1**, the stride (distance in bytes between two elements) is loaded into **r2**, and the number of vector elements is loaded into **r3**, then the parallel loop can be written as:

```
     vp     r0 = r3, L2   # allocate VPs, r0 = NVP
     set    r6 = 0        # r4 = partial sum in each VP
     dup    r2 = r2       # r2 = stride
     dup    r3 = r1, r2   # r3 = element address in each VP
     mul    r4 = r2, r0   # r5 = stride * NVP
L1:
     ld     r5 = [r3]     # r6 = loaded value
     add    r3 = r3, r4   # advance pointer in each VP
     add    r6 = r6, r5   # accumulate partial sum in each VP
     loop   L1
     sum    r7 = r4       # r7 = reduced partial sums
L2:
```



The LOOP Instruction

The **loop** instruction is used for executing a counter-controlled loop. It does the following:

```
if (VL > NVP + VPID) { VL = VL – NVP; jump label; }
else VL = 0;
```

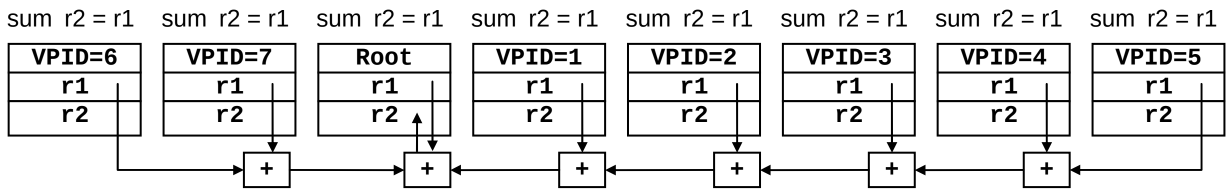
Suppose VL = 10 and NVP = 4. After executing the first iteration, VL becomes 6 and the loop instruction jumps to label to start the second iteration in all virtual processors. After executing the second iteration, VL becomes 2 and the loop instruction starts a third iteration in the first two virtual processors (VPID = 0 and 1), while the loop terminates (VL = 0) in the last two virtual processors (VPID = 2 and 3). Therefore, three iterations are executed in the first two virtual processors (VPID = 0 and 1) and two iterations are executed in the last two virtual processors (VPID = 2 and 3). This is equivalent to executing ten iterations if the loop is executed only by the root thread. The **loop** instruction guarantees that VL = 0 in all virtual processors when the loop terminates.

Reduction Instructions

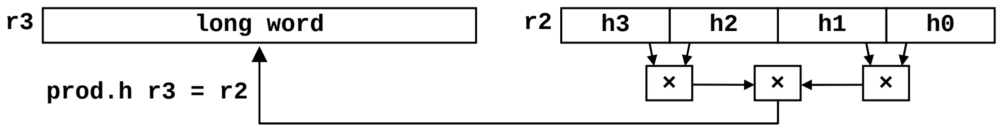
The **`sum`** instruction is a reduction instruction that initiates the transfer and reduction of a source register **`rs`** from all VPs into a destination register **`rd`** at the root VP. It acts as a send and injects a token into the network when issued by any VP. The injected token carries the opcode (**`sum`**), the root id, **`root`**, the source register number **`rs`**, the data **`value`**, and the initial **`count`** = **`1`**. Tokens injected into the network can be merged with other tokens carrying the same reduction opcode, the same root id, and the same register number. When merged, the token **`values`** are reduced according to the opcode, and the **`counts`** are added. At the root, the **`sum`** instruction acts as a blocking-receive. After receiving a merged token holding **`count=NVP`**, register **`rd`** is updated and instruction execution is resumed. The exact implementation can vary. One choice is to have reduction implemented in the network itself. Another possibility is to use binary tree reduction in the VP threads.

```
sum rd = rs
```

```
token = <op = sum, root, rs, value, count>
```



Reduction is also used to reduce the packed elements of a source register. The **`.f`** extension specifies the packed register format. For example, **`prod.h r3 = r2`** computes the product of half words packed in **`r2`** and stores the result in **`r3`**. This is useful even if only one thread is executing the reduction instruction. The result is a long word, regardless of the source register packed format.



In general, a reduction instruction is computed at two levels. First, each VP thread computes the reduction operation on the packed elements of a source register, if a packed format is specified. Then all the partial results are reduced into one final result at the root VP.

The list of reduction instructions include:

```
sum.f     rd = rs        # reduced sum
min.f     rd = rs        # reduced minimum
max.f     rd = rs        # reduced maximum
prod.f    rd = rs        # reduced product
sumu.f    rd = rs        # reduced sum (unsigned)
minu.f    rd = rs        # reduced minimum (unsigned)
maxu.f    rd = rs        # reduced maximum (unsigned)
produ.f   rd = rs        # reduced product (unsigned)
```

Reduction instructions apply to integer as well as to floating-point registers. The last four unsigned reduction instructions apply only to integer registers.

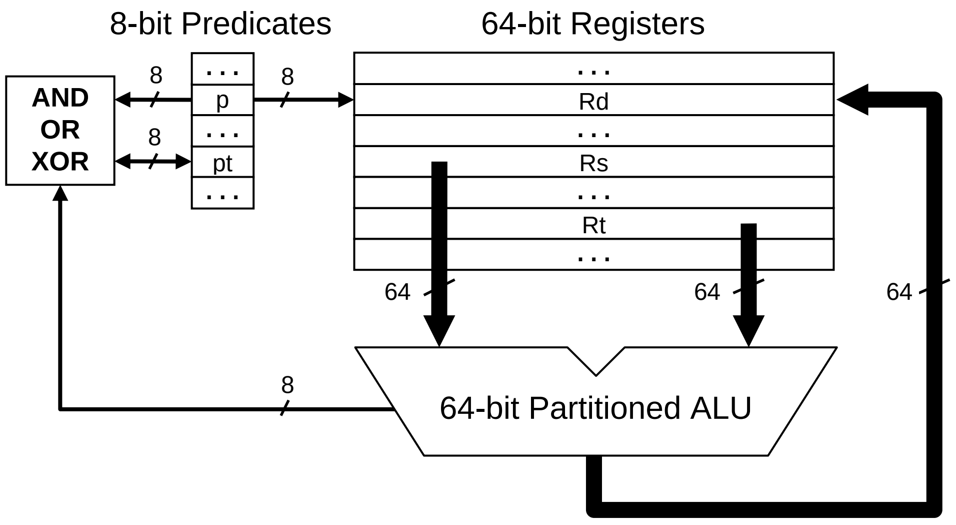
Predication

All instructions specify a qualifying predicate register. The general syntax of an integer instruction is given below, where `(p)` is the qualifying predicate, `op` is the operation, and `.f` represents the register format (with `.l` as a default value).

```
(p)   op.f     rd = rs, rt|im8
```

Because there are eight packed byte elements per 64-bit register, each predicate register `(p)` consists of 8 bits, with one bit per byte. The 8-bit predicate register `(p)` controls the writing of the individual bytes in register `rd`. If all the bits of predicate `(p)` are zeros then there is no need to issue the instruction for execution. The instruction can be dropped.

The following diagram depicts the operation of a typical integer ALU instruction. In addition to producing a 64-bit result, an 8-bit predicate value controls the writing of the packed bytes in destination register `Rd`.



Operation of a typical integer ALU instruction

Compare instructions produce a predicate value in register `pt`. The result of a compare instruction can be AND/OR/XOR-ed with a qualifying predicate `(p)`, as illustrated above.

When the qualifying predicate `(p)` is not specified, it is assumed to be `p0`, which is always `true`. It is reserved for non-conditional instructions.

Compare Instructions

There are only eight predicate registers, labeled **`p0-p7`**. Their complements are **`c0-c7`**. Either a predicate register or its complement can be used as a qualifying predicate in predicated instructions. **`p0`** is always true and all its bits are hardwired to 1's. It need not be specified in unconditional instructions.

Ten integer compare instructions are defined: **`eq`** (equal), **`ne`** (not equal), **`lt`** (less than), **`ltu`** (less than unsigned), **`le`** (less or equal), **`leu`** (less or equal unsigned), **`gt`** (greater than), **`gtu`** (greater than unsigned), **`ge`** (greater or equal), and **`geu`** (greater or equal unsigned). The syntax of a compare instruction (for example, **`eq`**) is given below:

```
(p)  eq.f   pt = rs, rt|im
```

It means the following:

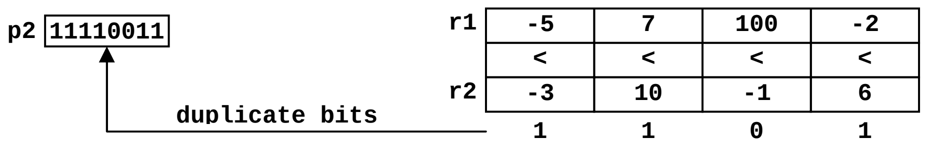
```
pt = (p) AND (rs == rt|im)    # bitwise AND with (p)
```

An integer compare instruction can use one of the following **`.f`** register formats extension: **`.b`** (packed bytes), **`.h`** (packed half words), **`.w`** (packed words), or **`.l`** (long word).

If the packed bytes format is used, eight arbitrarily different bits can be produced in **`pt`**. If the long word format is used, the result eight predicate bits will be all identical (either all zeros or all ones). If the register format is not specified, it defaults to long word (**`.l`**).

For example, the following **`lt.h`** instruction computes the less-than operation in parallel on the four packed half word elements. Since we have fixed the size of a predicate register to eight bits, each bit is duplicated to produce an eight bit result. The result is stored in **`p2`**.

```
lt.h   p2 = r1, r2      # p2 = r1<r2
```



The compare instruction can be predicated. The qualifying predicate is bitwise AND-ed with the corresponding bits of the compare instruction result. In the above example, suppose the value of **`p1=00111111`**, then the instruction **`(p1) lt.h p2 = r1, r2`** computes **`p2`** as: **`(p1) AND (r1<r2) = 00110011`**.

For floating-point, six compare instructions are defined: **`eq`** (equal), **`ne`** (not equal), **`lt`** (less than), **`le`** (less or equal), **`gt`** (greater than), and **`ge`** (greater or equal). Unsigned comparisons are used only for integer, but not for floating-point operands. Floating-point instructions can use either the **`.s`** (packed single-precision) or the **`.d`** (double-precision) register format. If the format is not specified for floating-point registers, it defaults to **`.d`**.

Example from the EEMBC Benchmark

Consider converting an RGB color array to CMYK for printing. The R, G, B, and the resulting C, M, Y, and K are 8-bit pixel color arrays. This conversion is one of the EEMBC Consumer Benchmark. It is described by the following loop:

```
for (i=0; i<n; i++) {
  c_val = 255 – R[i];
  m_val = 255 – G[i];
  y_val = 255 – B[i];
  K[i] = c_val;
  if (m_val < K[i]) K[i] = m_val;
  if (y_val < K[i]) K[i] = y_val;
  C[i] = c_val – K[i];
  M[i] = m_val – K[i];
  Y[i] = y_val – K[i];
}
```

The above loop can be parallelized as follows. Predication is used to translate the nested if-statements.

```
# Assume r0 = n,  r1 = &R, r2 = &G, r3 = &B
# Assume r4 = &C, r5 = &M, r6 = &Y, r7 = &K
# n is a multiple of 8, addresses are aligned on 8-bytes

     srl      r8  = r0, 3     # r8 = n/8, 8 bytes per register
     vp       r9  = r8, L2    # allocate virtual processors
L1:  lv       r10 = [r1+]     # r10 = 8 packed bytes of R[]
     lv       r11 = [r2+]     # r11 = 8 packed bytes of G[]
     lv       r12 = [r3+]     # r12 = 8 packed bytes of B[]
     subfu.b  r10 = r10, 255  # r10 = c_val = 255 - R[i]
     subfu.b  r11 = r11, 255  # r11 = m_val = 255 - G[i]
     subfu.b  r12 = r12, 255  # r12 = y_val = 255 - B[i]
     mov      r14 = r10       # r14 = compute unsigned minimum
     ltu.b    p2  = r11, r14  #       of (r10, r11, r12)
(p2) mov      r14 = r11
     ltu.b    p2  = r12, r14
(p2) mov      r14 = r12       # r14 = 8 packed bytes of K[]
     subfu.b  r10 = r14, r10  # r10 = 8 packed bytes of C[]
     subfu.b  r11 = r14, r11  # r11 = 8 packed bytes of M[]
     subfu.b  r12 = r14, r12  # r12 = 8 packed bytes of Y[]
     sv       [r4+] = r10     # store 8 packed bytes of C[]
     sv       [r5+] = r11     # store 8 packed bytes of M[]
     sv       [r6+] = r12     # store 8 packed bytes of Y[]
     sv       [r7+] = r13     # store 8 packed bytes of K[]
     loop     L1
L2:
```

Translating If-statements

Predication works very nicely when translating an if-statement nested inside a loop. The Boolean expressions can include logical AND/OR operations. The following are examples:

```
# Evaluation of logical AND
if (ch >= 'A' && ch <= 'Z') ch = ch + 32;

# Assume 8 characters (ch) are packed in register r8

      ge.b      p1 = r8, 'A'    # p1 = (ch >= 'A')
(p1)  and.le.b  p1 = r8, 'Z'    # p1 = (p1) AND (ch <= 'Z')
(p1)  add.b     r8 = r8, 32     # if (p1) ch = ch + 32

# Evaluation of logical OR
if (ch < 'A' || ch > 'Z') ch = '*';

      lt.b      p2 = r8, 'A'    # p2 = (ch < 'A')
(p2)  or.gt.b   p2 = r8, 'Z'    # p2 = (p2) OR (ch > 'Z')
(p2)  set.b     r8 = '*'        # if (p2) ch = '*'
```

Compare instructions can be prefixed with a logical AND/OR operations as shown in the above two examples. The qualifying predicate is bitwise AND/OR-ed with the comparison result. If the prefix is not specified, it defaults to AND.

```
# Nested if-else statement
if (ch >= '0' && ch <= '9') d = ch – '0';
else if (ch >= 'A' && ch <= 'F') d = ch – 55;
else if (ch >= 'a' && ch <= 'f') d = ch – 87;
else d = 0;

# Assume 8 characters (ch) are packed in register r8
# Assume 8 digits (d) are packed in register r9

      ge.b      p2 = r8, '0'    # p2 = (ch >= '0')
(p2)  le.b      p2 = r8, '9'    # p2 = (p2) & (ch <= '9')
(p2)  add.b     r9 = r8, -48    # if (p2) d = ch – '0'
(c2)  ge.b      p3 = r8, 'A'    # p3 = (~p2) & (ch >= 'A')
(p3)  le.b      p3 = r8, 'F'    # p3 = (p3) & (ch <= 'F')
(p3)  add.b     r9 = r8, -55    # if (p3) d = ch – 55
      orp       p2 = p2, p3     # p2 = (p2) | (p3)
(c2)  ge.b      p3 = r8, 'a'    # p3 = (~p2) & (ch >= 'a')
(p3)  le.b      p3 = r8, 'f'    # p3 = (p3) & (ch <= 'f')
(p3)  add.b     r9  = r8, -87   # if (p3) d = ch – 87
      orp       p2 = p2, p3     # p2 = (p2) | (p3)
(c2)  set.b     r9  = 0         # if (p3) r9 = 0
```

The `orp` instruction is used to OR predicate registers. We can also define `andp` and `xorp` instructions that operate on predicate registers.

```
# If conditions are mutually exclusive
# Then parallel comparisons – same above example

if (ch >= '0' && ch <= '9') d = ch – '0'
else if (ch >= 'A' && ch <= 'F') d = ch – 55
else if (ch >= 'a' && ch <= 'f') d = ch – 87
else d = 0

# Assume 8 characters (ch) are packed in register r8
# Assume 8 digits (d) are packed in register r9

      ge.b    p1 = r8, '0'      # p1 = (ch >= '0')
      ge.b    p2 = r8, 'A'      # p2 = (ch >= 'A')
      ge.b    p3 = r8, 'a'      # p3 = (ch >= 'a')

(p1)  le.b    p1 = r8, '9'      # p1 = (p1) AND (ch <= '9')
(p2)  le.b    p2 = r8, 'F'      # p2 = (p2) AND (ch <= 'F')
(p3)  le.b    p3 = r8, 'f'      # p3 = (p3) AND (ch <= 'f')

      set     r9 = 0            # default case
(p1)  add.b   r9 = r8, -48      # if (p1) d = ch – '0'
(p2)  add.b   r9 = r8, -55      # if (p2) d = ch – 55
(p3)  add.b   r9 = r8, -87      # if (p3) d = ch – 87
```

Comments:

Code exposing ILP and DLP

First 3 instructions can potentially execute in parallel
Next 3 depend on first 3 and can also execute in parallel
8 bytes within a single register also operate in parallel

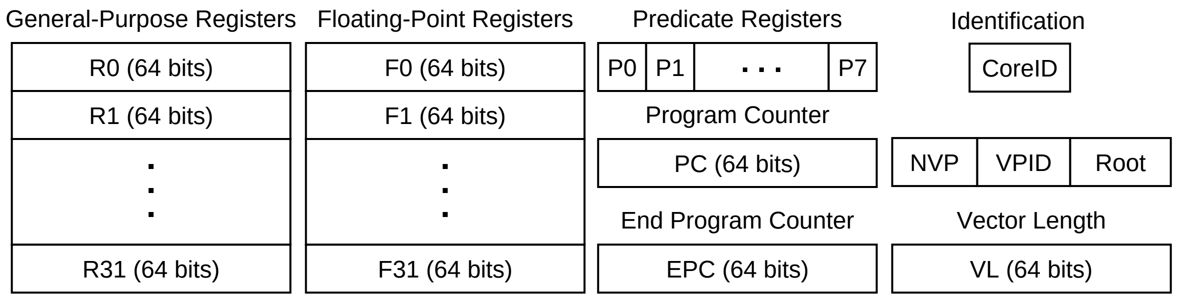
```
# Setting and clearing a predicate register

      eq      p2 = r0, r0       # setting all bits in p2
      ne      p3 = r0, r0       # clearing all bits in p3
```

Execution Environment

Each VP (virtual processor) can address the following registers:

- 32 × 64-bit general-purpose registers: *r*0 – *r*31
  - *r*0 is a normal register (not hardwired to zero).
  - A general-purpose register can also pack 8 bytes, 4 half words, or 2 words
- 32 × 64-bit floating-point registers: *f*0 – *f*31
  - A floating-point register can also pack 2 single-precision floats
- 8 × 8-bit predicate registers: *p*0 – *p*7
  - Almost all instructions are predicated. There can be few exceptions.
  - Each predicate register is 8 bits, with one bit per byte in the general-purpose register.
  - *p*0 is always true. All bits of *p*0 are hardwired to one, for unconditional execution.
  - *p*1 is the overflow/carry/saturation register, generated by many instructions.
- PC and EPC
  - EPC marks the last instruction address in a thread.
  - Thread execution terminates when PC = EPC, and its register context is freed.
  - The instruction at address EPC is not executed
- Identification
  - CoreID is the ID of the core running this thread.
  - Root is the ID of the core running the root thread.
  - NVP is the number of virtual processors (or cores) running the thread.
  - VPID is the ID of this virtual processor (or thread).
- Vector Length (VL) register
  - Used by some instructions to control loop execution
- Not complete: feel free to add additional registers if needed



Execution environment registers

Immediate Constants

Integer instructions can specify an 8-bit immediate constant for the second source operand, in place of a second source register. Two instruction formats will result: R-R (both source operands are registers) and R-I (second source operand is an immediate constant). For simplicity the same mnemonic is used for both the R-R and R-I formats. Consider again the **`add`** instruction, there are two such instructions, where **`.f`** represents the register format:

```
add.f    rd = rs, rt     # R-R format
add.f    rd = rs, im8    # R-I format, 8-bit immediate
```

The 8-bit immediate constant is sign-extended for all integer instructions. When adding an immediate constant to a packed register format, the immediate constant is *replicated to all packed elements*. The following diagram shows an example:

r1	Half word 3	Half word 2	Half word 1	Half word 0
	+ 1	+ 1	+ 1	+ 1
	=	=	=	=
r2	Half word 3	Half word 2	Half word 1	Half word 0

`add.h r2 = r1, 1 # add 1 to each packed element`

The SET Instruction

The **`set`** instruction is used to initialize a destination register **`rd`** with a 16-bit immediate constant. It has the following syntax, where **`.f`** is the register format:

```
(p)    set.f    rd = imm16
```

The immediate constant is replicated for packed bytes (**`.b`** extension), packed half words (**`.h`** extension), and packed words (**`.w`** extension). To set packed bytes, only the least significant 8-bit of the immediate constant is replicated. To set packed words, the immediate constant is sign-extended to 32 bits and replicated. To set a long word, the immediate constant is sign-extended to 64 bits. The following are examples:

`set.b r3 = 5`	**r3**	5	5	5	5	5	5	5	5
`set.h r3 = -713`	**r3**	-713		-713		-713		-713	
`set.w r3 = -713`	**r3**	-713				-713			
`set.l r3 = -713`	**r3**	-713							

Using Predication to Control Register Writing

Predication is used to control the writing of destination register **`rd`**. Since a predicate register consists of 8 bits, each bit is used to control the writing of one corresponding byte. In the following example, if **`p1 = 11110000`** then only bytes 4, 5, 6, and 7 can be written in **`r3`** (shown in red). Note that predicate values should be generated properly for the desired register format. Otherwise, bytes can be incorrectly written in a destination register.

Initial value of r3	**r3**	0x12	0x34	0x56	0x78	0x90	0xab	0xcd	0xef
`(p1) set.b r3 = 0x37`	**r3**	0x37	0x37	0x37	0x37	0x90	0xab	0xcd	0xef
`(p1) set.h r3 = 0xfd37`	**r3**	0xfd37		0xfd37		0x90ab		0xcdef	
`(p1) set.w r3 = 0xfd37`	**r3**	0xffffd37				0x90abcdef			

Integer Instruction Formats

Three instruction formats are defined for R-R, R-I, and SET instructions.

R-R Format

p^3	c	op^4	f^2	rd^5	rs^5	rt^5	opx^7

R-I Format

p^3	c	op^4	f^2	rd^5	rs^5	imm8	opx^4

SET Format

p^3	c	op^4	f^2	rd^5	▨	imm16

- **`p`** = qualifying predicate = (**`p0`** thru **`p7`**)
  - Qualifying predicate **`(p)`** is used to control the writing of each byte in register **`rd`**
  - If **`p = 00000000`** then there is no need to execute instruction
- **`c`** = Complement of a qualifying predicate = (**`c0`** thru **`c7`**)
- **`op`** = major opcode
  - Defines the instruction format
  - One major opcode is used for all R-R integer instructions
  - One major opcode is used for all R-I integer instructions
  - One major opcode is used for all Floating-point instructions
- **`f`** = Register format
  - **`.b`** (packed bytes), **`.h`** (packed half words), **`.w`** (packed words), **`.l`** (long word)
- **`rd`** = Destination register (always written)
- **`rs`** = First source register (always read)
- **`rt`** = Second source register (always read)
- **`opx`** = Opcode extension
  - 7-bit opcode extension for R-R format defines up to 128 functions per major opcode
  - 4-bit opcode extension for R-I format defines up to 16 functions per major opcode
- **`imm8`** = 8-bit signed immediate constant used in R-I instructions
  - Immediate constant is sign-extended and/or replicated according to register format
- **`imm16`** = 16-bit signed immediate constant used in SET instructions
  - Immediate constant is sign-extended and/or replicated according to register format

Integer Instructions

Integer instructions operate on 64-bit general-purpose registers. They include arithmetic, comparison, logical, and shift instructions. They can operate on 64-bit long integers or narrower packed data. They can execute conditionally based on the content of a qualifying predicate register.

Integer Addition

There are two integer addition instructions: **`add`** and **`addu`**. Signed addition may overflow indicating that the result is out of the signed range. Similarly, unsigned addition may produce a carry, indicating that the result exceeded the maximum unsigned value. In both cases, modulo-arithmetic is used and the result is identical, but can be out-of-range. The **`p1`** register is used to store the overflow/carry flags for signed/unsigned addition.

Saturation

Sometimes, it is desirable to produce saturated (rather than modulo) values when results are out-of-range. Signed saturation clamps the result to the maximum positive or minimum negative value that can be represented. Unsigned saturation clamps the result to the maximum unsigned value, or to zero (in case of unsigned subtraction).

We can have a saturation option by adding the letter (**`s`**) after the instruction mnemonic. This will result in the following combinations of addition instructions:

```
(p) add.f     rd = rs, rt|im8  # add signed   (modulo)
(p) addu.f    rd = rs, rt|im8  # add unsigned (modulo)
(p) adds.f    rd = rs, rt|im8  # add with signed saturation
(p) addus.f   rd = rs, rt|im8  # add with unsigned saturation
```

An example of signed addition on packed bytes with and without saturation is shown below. In the first example, modulo-addition is used and the result overflows in bytes 2 and 3. The **`p1`** predicate register captures the eight overflow flags. In the second example, saturated-addition is used and the result is saturated in bytes 2 and 3. Observe that bytes 6 and 7 are not saturated, although they contain the maximum positive and minimum negative byte values.

r1	-127	1	6	8	-95	100	5	-3
	+	+	+	+	+	+	+	+
r2	-1	126	13	7	-63	50	-4	-2
	=	=	=	=	=	=	=	=
r3	-128	127	19	15	98	-106	1	-5

`add.b r3=r1,r2 # p1 = 00001100`

r1	-127	1	6	8	-95	100	5	-3
	+	+	+	+	+	+	+	+
r2	-1	126	13	7	-63	50	-4	-2
	=	=	=	=	=	=	=	=
r3	-128	127	19	15	-128	127	1	-5

`adds.b r3=r1,r2 # p1 = 00001100`

The saturation option applies to only few integer instructions (addition, subtraction, multiplication, and packing). The remaining integer instructions do not saturate.

As an alterative approach, it is also possible to define a **`sat`** (saturate) instruction that clamps a signed integer to its maximum positive or minimum negative value, depending on its sign:

```
(p)  sat.f    rd = rs           # if (p) rd = saturate(rs)
```

For the above example, **`adds.b r3 = r1, r2`** is equivalent to:

```
     add.b    r3 = r1, r2       # p1 = 00001100
(p1) sat.b    r3 = r1           # bytes 2,3 of r3 are saturated
```

Integer Subtraction

One way to define integer subtraction is: **`rd = rs – rt|im8`**. A problem with this definition is that to obtain the negation of register **`rt`**, one must use a register **`rs`** with a zero value. However, no general-purpose register is hardwired to zero. **`r0`** is a normal register that can be updated. Another problem is that there is no use for the 8-bit immediate constant. To compute: **`rd = rs – im8`**, one can use the **`add`** instruction with a negative immediate.

A better way to define subtraction is: **`rd = -rs + rt|im8`**. This is called *subtract from*. To obtain the negative of register **`rs`**, a zero immediate can be used. The use of an immediate constant is also useful when adding it to the negative of register **`rs`**.

Therefore, four *subtract-from* instructions are defined for signed/unsigned operands, and with or without saturation:

```
(p) subf.f    rd = rs, rt|im8  # rd = -rs + rt|im8
(p) subfu.f   rd = rs, rt|im8  # rd = -rs + rt|im8
(p) subfs.f   rd = rs, rt|im8  # with signed saturation
(p) subfus.f  rd = rs, rt|im8  # with unsigned saturation
```

To obtain the negation of register **`rs`**, write: **`subf rd = rs, 0  # rd = -rs + 0`**

As with addition, the **`subf`** and **`subfu`** produce identical results, because modulo-arithmetic is used. However, the **`subf`** instruction produces eight *overflow* flags, while the **`subfu`** produces eight *borrow* flags for unsigned subtraction. The result overflow/borrow flags are stored in predicate register **`p1`**. The **`subfs`** and **`subfus`** might produce different saturated results. If a borrow is detected with unsigned saturation, the result is saturated to zero, because it cannot be negative.

To simplify assembly-language programming, the following pseudo-instructions are defined. The pseudo-instruction **`sub r3 = r1, r2`** computes **`r3 = r1 – r2`**. It is equivalent to the real instruction **`subf r3 = r2, r1`** where the order of the source operands is exchanged.

Pseudo Instruction	Equivalent Real Instruction
`(p)  neg.f     r2 = r1`	`(p)  subf.f    r2 = r1, 0`
`(p)  sub.f     r3 = r1, r2`	`(p)  subf.f    r3 = r2, r1`
`(p)  subu.f    r3 = r1, r2`	`(p)  subfu.f   r3 = r2, r1`
`(p)  subs.f    r3 = r1, r2`	`(p)  subfs.f   r3 = r2, r1`
`(p)  subus.f   r3 = r1, r2`	`(p)  subfus.f  r3 = r2, r1`

Pack

The **pack** instruction is used to pack two source registers **rs** and **rt** into a destination register **rd**. Long words are packed into words, words are packed into half words, and half words are packed into bytes. The packed elements of **rs** go into the upper half or **rd**. The packed elements of **rt** go into the lower half. The syntax of the **pack** instruction is given below:

```
(p)   pack.f    rd = rs, rt      # pack rs, rt into rd
```

Three integer register formats are supported for the packed register **rd**: **.w**, **.h**, and **.b**.

When packing, signed and unsigned saturation options can saturate the truncated integer operands. Out-of-range values are saturated to the maximum or minimum values that can be represented. Two saturation options can be used with the **pack** instruction:

```
(p)   packs.f   rd = rs, rt      # pack with signed saturation
(p)   packus.f  rd = rs, rt      # pack with unsigned saturation
```

If saturation is detected, the **p1** predicate bits are set for the corresponding saturated bytes. The following example illustrates the packing of words into packed half words with signed and unsigned saturation. Saturated values are shown in red. **p1** is equal to **00111100** after executing **packs.h**, and it is equal to **11001100** after executing **packus.h**.

Assume value of r1	r1	-1		50000	
Assume value of r2	r2	-33000		32767	
packs.h r3 = r1, r2	r3	-1	32767	-32768	32767
packus.h r3 = r1, r2	r3	65535	50000	65535	32767

Unpack

The **unpk** instruction is used to unpack the low order 32 bits of a source register **rs** into a 64-bit destination register **rd**. 4 bytes are unpacked into 4 half words, 2 half words are unpacked into 2 words, and a word is unpacked into a long word. The **unpkh** instruction is used to unpack the high order 32 bits. Packed elements are sign- or zero-extended, depending on the option. The four variations of the **unpk** instruction are given below:

```
(p)   unpk.f    rd = rs    # unpack low  32 bits sign-extended
(p)   unpku.f   rd = rs    # unpack low  32 bits zero-extended
(p)   unpkh.f   rd = rs    # unpack high 32 bits sign-extended
(p)   unpkhu.f  rd = rs    # unpack high 32 bits zero-extended
```

Three register formats are supported by the **unpk** instruction: **.h**, **.w**, and **.l**, which specify the format of destination register **rd**. No saturation can occur and the **p1** predicate register is not affected.

Integer Multiplication

In general, integer multiplication takes two operands of size *n* bits and produces a product of size 2*n* bits. Because the destination register is only *n* bits in size, the **mul** instruction is defined here to produce only the low-order *n* bits of the multiplication. There is no need to distinguish between signed and unsigned multiplication because the lower *n* bits of the product are the same in both cases. The difference is in the upper *n* bits, which are not stored.

```
(p)  mul.f    rd = rs, rt      # rd = rs * rt
```

Restricting the product size to *n* bits is not a limitation. If a 2*n* bit product is desired, the operands can be expanded from *n* to 2*n* bits before multiplication, using the **unpk** (unpack) instruction. Sign/zero extensions are used to unpack signed/unsigned integer operands.

Assume value of r1	**r1**	**unused**	**unused**	**-5000**	**1234**
Assume value of r2	**r2**	**unused**	**unused**	**378**	**100**
unpk.w r3 = r1	**r3**	**-5000**		**1234**	
unpk.w r4 = r2	**r4**	**378**		**100**	
mul.w r5 = r3, r4	**r5**	**1890000**		**123400**	

Signed/unsigned saturation options can be defined for the **mul** instruction, which affect the **p1** predicate. Saturation options are useful if no unpacking is done before multiplication.

```
(p)  muls.f   rd = rs, rt      # with signed saturation
(p)  mulus.f  rd = rs, rt      # with unsigned saturation
```

Integer Division

There are two integer divide instructions **div** and **mod**, which produce the quotient and remainder. Both instructions take two *n*-bit operands and produce an *n*-bit quotient and an *n*-bit remainder. The division can be signed or unsigned and can operate on packed elements within a register according to the register format. There is no saturation option with integer division because the quotient and the remainder do not saturate.

```
(p)  div.f   rd = rs, rt      # Signed quotient
(p)  mod.f   rd = rs, rt      # Signed remainder
(p)  divu.f  rd = rs, rt      # Unsigned quotient
(p)  modu.f  rd = rs, rt      # Unsigned remainder
```

The division hardware can produce the quotient and remainder simultaneously. However, since there is only one destination register in the instruction format and to avoid using special-purpose HI and LO registers, the **mod** instruction is introduced in addition to **div**. The **mod** instruction will do integer division identical to **div**. Sometimes, a program needs either the quotient or the remainder, but not both. In this case, the proper instruction is used and the division is done once. However, if a program requires both results and uses both instructions in sequence, the division hardware can save the last division operands and result in internal registers to avoid repeating the same operation and wasting cycles.

Logical Instructions

Eight R-R logical instructions are defined as follows:

```
(p) and      rd = rs, rt      # rd =   rs & rt
(p) andc     rd = rs, rt      # rd =  ~rs & rt
(p) or       rd = rs, rt      # rd =   rs | rt
(p) orc      rd = rs, rt      # rd =  ~rs | rt
(p) xor      rd = rs, rt      # rd =   rs ^ rt
(p) nand     rd = rs, rt      # rd = ~(rs & rt)
(p) nor      rd = rs, rt      # rd = ~(rs | rt)
(p) xnor     rd = rs, rt      # rd = ~(rs ^ rt)
```

The register format (**.f** extension) need not be specified for R-R logical instructions, because these instructions operate on bits that are not affected by the register format.

Four R-I logical instructions are defined as follows:

```
(p) and.f    rd = rs, im8     # rd =  rs & im8
(p) andc.f   rd = rs, im8     # rd = ~rs & im8
(p) or.f     rd = rs, im8     # rd =  rs | im8
(p) xor.f    rd = rs, im8     # rd =  rs ^ im8
```

The register format (**.f** extension) should be used for R-I logical instructions. It indicates how the immediate constant should be replicated for the packed elements.

Logical instructions do not produce overflow or carry bits, and hence do not modify the **p1** predicate register.

Shift and Rotate Instructions

Three shift instructions and one rotate are defined as follows:

```
(p) sll.f    rd = rs, rt|im   # shift left logical
(p) srl.f    rd = rs, rt|im   # shift right logical
(p) sra.f    rd = rs, rt|im   # shift right arithmetic
(p) rol.f    rd = rs, rt|im   # rotate left
```

Shifting and rotation differ depending on the register format. If a packed register format is specified, then each packed element is shifted or rotated. The shift/rotate amount is either an immediate constant **im** or a variable stored in register **rt**. For packed byte elements, only three bits are required to specify the shift/rotate amount, while the long integer format requires six bits. There is only one rotate left (**rol**) instruction. There is no rotate right. To rotate right a 64-bit long integer by 20 bits, it can be rotated left by 44 = 64 – 20 bits.

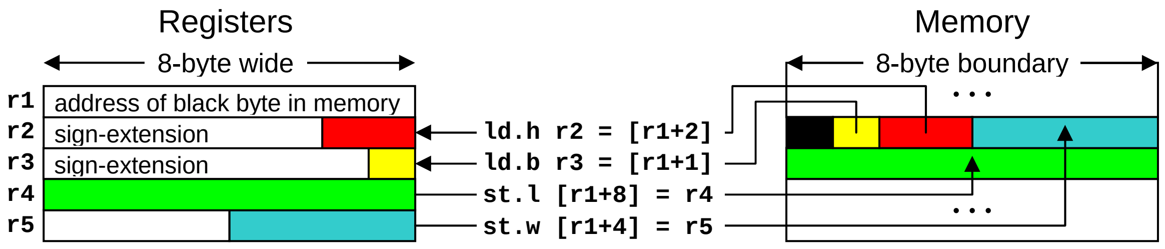
Scalar Load and Store

The load and store instructions are defined as follows:

```
(p)  ld.f   rd = [rs+im]          # if (p!=0) rd = memory[rs+im]
(p)  st.f   [rs+im] = rt          # if (p!=0) memory[rs+im] = rt
```

The **.f** extension can be **.b** (load/store byte), **.h** (half word), **.w** (word), or **.l** (long word). **ld.b** loads one byte from memory into the least-significant byte of a destination register **rd**, **ld.h** loads one half word, **ld.w** loads one word, and **ld.l** loads a long word (which might pack 8 bytes, 4 half words, or 2 words). Sign-extension fills the upper-part of a register.

Similarly, **st.b** stores the least significant byte of a source register **rt**, **st.h** stores the lower half word, **st.w** stores the lower word, and **st.l** stores the complete register (which might pack 8 bytes, 4 half words, or 2 words). If the **.f** extension is not specified, it is assumed to be **.l** for integer loads and stores.



For floating-point, the **.f** extension can be **.s** (single-precision) or **.d** (double-precision). Destination register **rd** is replaced by **fd**, and source register **rt** is replaced by **ft**.

Memory Alignment

Memory alignment is a requirement to simplify the hardware implementation. Long words should be aligned on an 8-byte boundary, words should be aligned on a 4-byte boundary, and half-words should be aligned on a 2-byte boundary. Bytes need not be aligned.

Addressing Mode

Only one addressing mode is supported by the load and store instructions: *base-displacement addressing*. The base address is stored in register **rs**, and the displacement is the immediate constant. The effective address is computed as follows:

```
Effective Address = Reg(rs) + sign-extend(immediate)
```

Load and Store Format

The load and store instructions use slightly different formats. The load instruction specifies register **`rd`** as the destination register, which is written by the load instruction, while the store instruction specifies register **`rt`** as a source register read by the store instruction and written in memory. Both instructions use a 12-bit signed immediate constant as a displacement, but the immediate is distributed in the store instruction format.

Load Format

p^3	op^5	f^2	rd^5	rs^5	Imm^{12}

Store Format

p^3	op^5	f^2	Imm^5	rs^5	rt^5	Imm^7

Control Flow Instructions

The **`jump`** and **`call`** instructions are defined as follows:

```
(p)  jump   label              # if (p!=0) jump to label
(p)  call   label              # if (p!=0) call procedure
(p)  jump   rs                 # if (p!=0) jump register
(p)  call   rs                 # if (p!=0) call register
```

If any bit of **`(p)`** is set, the **`jump`** or **`call`** instruction will occur. If all bits of **`(p)`** are clear, the **`jump`** or **`call`** instruction will be dropped. If **`(p)`** is not specified, it defaults to **`(p0)`**, and the **`jump`** or **`call`** will be unconditional.

The JUMP instruction format is used for the direct **`jump`** and **`call`** instructions. PC-relative addressing is used. The 24-bit immediate constant is sign-extended, shifted left by 2 bits, and added to the program counter to determine the address of the target instruction.

```
PC = PC + imm24 << 2
```

All instructions occupy 4 bytes in memory and are aligned on a 4-byte boundary. The lower 2 bits of PC are always zero. Therefore, **`imm24`** is simply added to the upper 62 bits of PC.

JUMP Format

p^3	op^5	imm24

The **`call`** instruction does a jump and saves the return address in register **`r31`**. This register is named the **`ra`** (return address) register.

The indirect **`jump`** and **`call`** instructions jump to the content of register **`rs`**. These two instructions use the R-R instruction format.

Floating-Point Instructions

Floating-point instructions can be designed to address a separate set of floating-point registers. Alternatively, it can address the same general-purpose registers. Each approach has its advantages and disadvantages. Addressing a separate set of registers doubles the number of registers that can be used by one thread, which might be considered a positive or negative point. It is a positive point if a program thread needs all these registers. However, it is a negative point if a program is dominated by integer instructions, which make little use of floating-point registers or vice-versa. It also takes more time to save and restore two register files (instead of one) on context switches. Furthermore, additional instructions are necessary to move data between the two register sets and to load/store floating-point registers. These instructions can be eliminated if a common register file is used. On the other hand, using a common register file places more pressure on the registers. More register ports are needed to issue multiple instructions in a given cycle. These ports can be reduced and distributed if two register files are used. There can be no register dependency between an integer and a floating-point instruction if separate register files are used, and hence can be issued in parallel. However, register dependencies must be checked if a common register file is used. In what follows, a separate set of floating-point registers **`f0–f31`** is assumed.

The following floating-point instructions are defined:

```
(p) add.f    fd = fs, ft     # if (p) fd = fs + ft
(p) sub.f    fd = fs, ft     # if (p) fd = fs - ft
(p) mul.f    fd = fs, ft     # if (p) fd = fs * ft
(p) div.f    fd = fs, ft     # if (p) fd = fs / ft
(p) mov.f    fd = ft         # if (p) fd = ft
(p) neg.f    fd = ft         # if (p) fd = -ft
(p) abs.f    fd = ft         # if (p) fd = abs(ft)
(p) rcp.f    fd = ft         # if (p) fd = 1/ft
(p) sqrt.f   fd = ft         # if (p) fd = sqrt(ft)
```

Where **`.f`** is the floating-point register format, which should be specified as either: **`.d`** or **`.s`**. Floating-point immediate constants cannot be specified as a second operand. They should be loaded from memory.

Conversion Instructions

The **`float`** instruction converts a long integer or a pair of integer words to their floating-point representation. It also transfers the converted data from a general-purpose register, **`rs`**, to a floating-point register, **`fd`**.

The **`round`**, **`trunc`**, **`floor`**, and **`ceil`** instructions convert a double-precision float or a pair of single-precision floats to their integer representation, using rounding to the nearest integer, truncation, or obtaining the floor or ceiling functions, which produce slightly different integer results. The converted data is transferred from a floating-point source register **`fs`** to a general-purpose register **`rd`**.

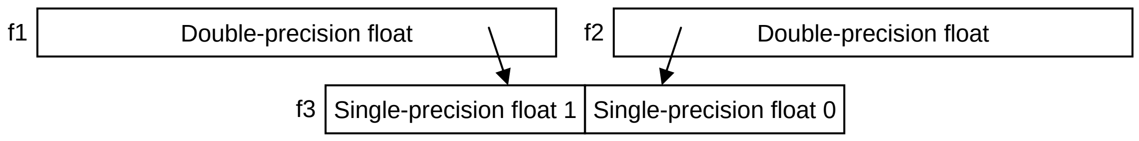
Only two register formats can be specified for all conversion instructions: either **`.d`** or **`.s`**.

```
(p) float.f   fd = rs          # fd = float(rs)
(p) round.f   rd = fs          # rd = round(fs)
(p) trunc.f   rd = fs          # rd = trunc(fs)
(p) floor.f   rd = fs          # rd = floor(fs)
(p) ceil.f    rd = fs          # rd = ceil(fs)
```

Floating-Point Pack

The **`pack.s`** instruction is used to pack two double-precision floating-point registers **`fs`** and **`ft`** into a packed single-precision destination register **`fd`**. Only the **`.s`** register format can be used to specify the format of destination register **`fd`**. The packed **`fs`** goes into the upper half or **`fd`**. The packed **`ft`** goes into the lower half.

```
(p)  pack.s   fd = fs, ft     # pack fs, ft into fd
```

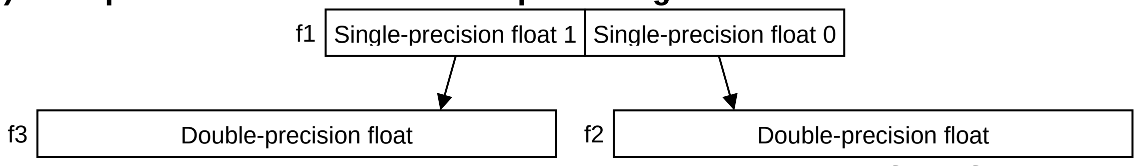


`pack.s f3 = f1, f2  # pack double-precision floats`

Floating-Point Unpack

The **`unpk.d`** instruction is used to unpack the low-order single-precision float of source register **`fs`** into a double-precision float in destination register **`fd`**. The **`unpkh.d`** instruction is used to unpack the high order single-precision float. Only the **`.d`** register format can be used to specify the format of destination register **`fd`**.

```
(p)  unpk.d   fd = fs   # unpack low  order float
(p)  unpkh.d  fd = fs   # unpack high order float
```



`unpkh.d f3 = f1` **`unpk.d f2 = f1`**