Lemberg

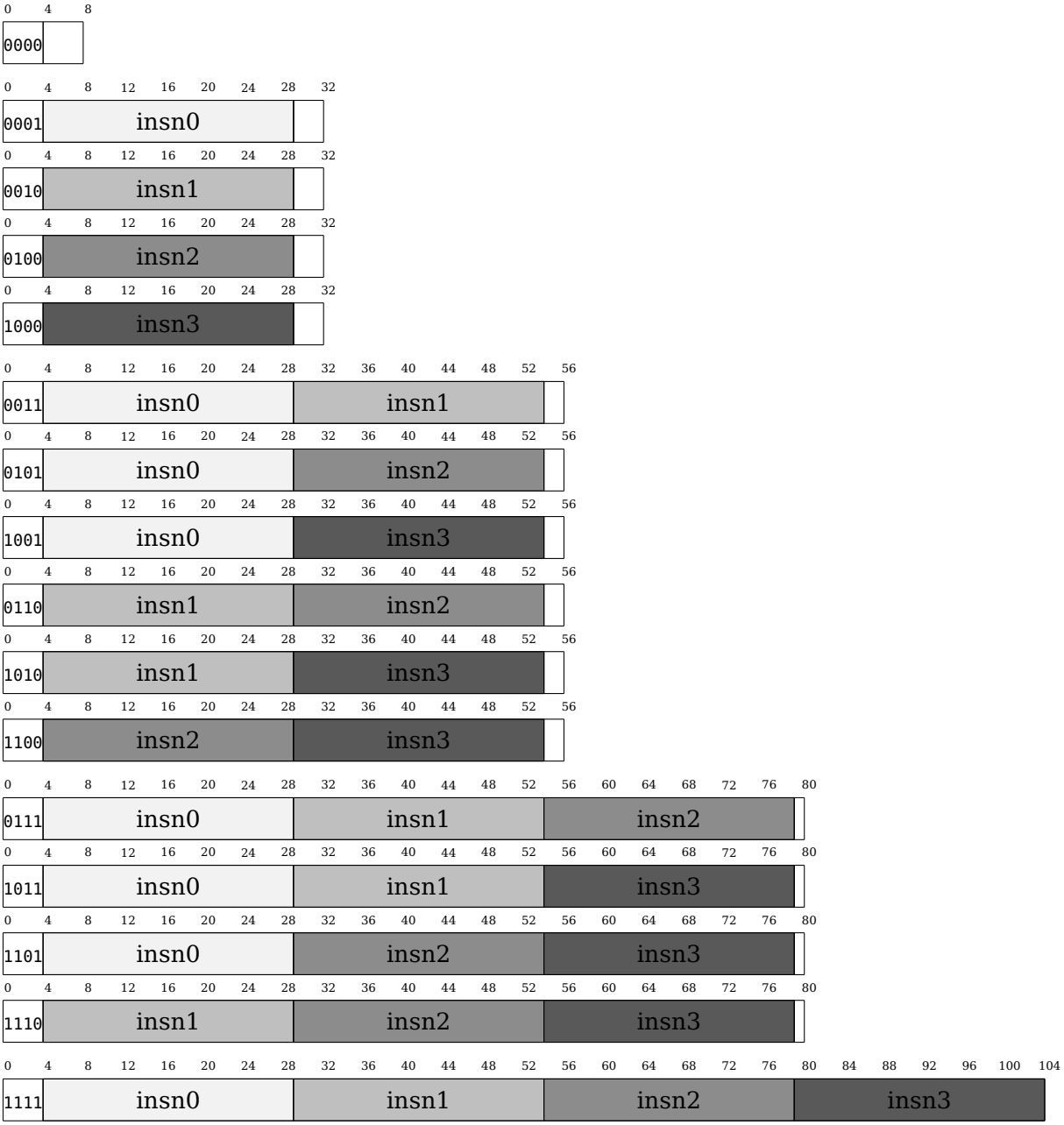
Wolfgang Puffitsch

– Why do you pronounce VLIW with an “F” at the end?
– Because I also pronounce Lviv that way.

Contents

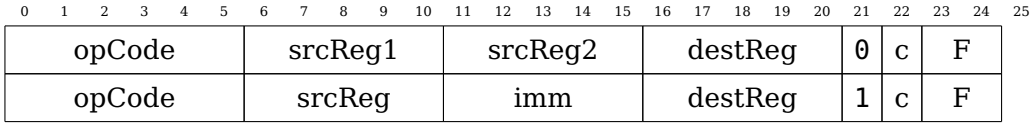
1 Opcode Formats

1.1 Bundle Formats



1.2 Instruction Formats

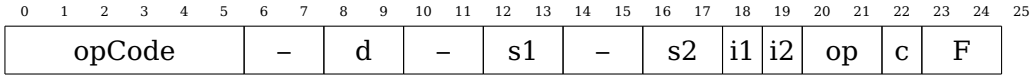
- Base format **B**:



0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23 24 25
opCode	srcReg1	srcReg2	destReg	`0`	c	F
opCode	srcReg	imm	destReg	`1`	c	F

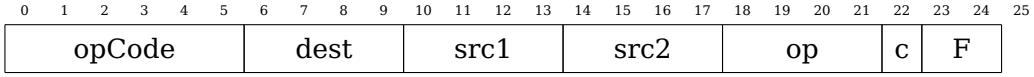
  For comparison and test operations, destReg refers to a condition flag.

- Flag combination format **C**:



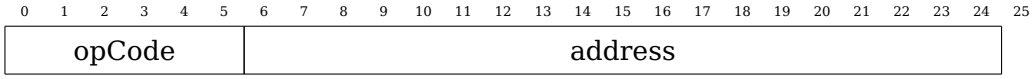
0 1 2 3 4 5	6 7	8 9	10 11	12 13	14 15	16 17	18	19	20 21	22	23 24 25
opCode	–	d	–	s1	–	s2	i1	i2	op	c	F

- Floating-point format **F**:



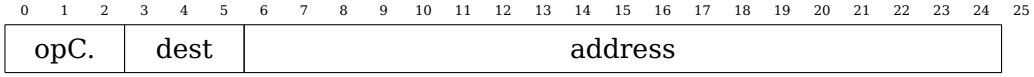
0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20 21	22	23 24 25
opCode	dest	src1	src2	op	c	F

- Global address format **G**:



0 1 2 3 4 5	6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
opCode	address

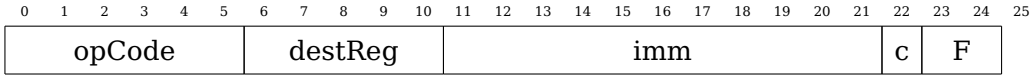
- Global address load format **H**:



0 1 2	3 4 5	6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
opC.	dest	address

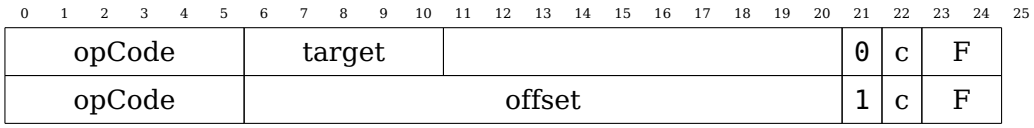
  dest values `000`-`011` address r0-r3, values `100`-`111` address r16-r18.

- Immediate load format **I**:



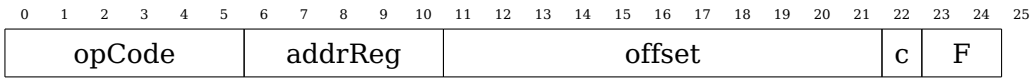
0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15 16 17 18 19 20 21	22	23 24 25
opCode	destReg	imm	c	F

- Jump format **J**:



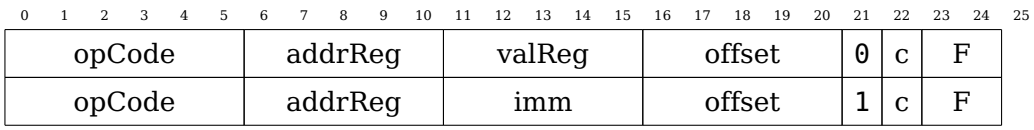
0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15 16 17 18 19 20	21	22	23 24 25
opCode	target		`0`	c	F
opCode	offset		`1`	c	F

- Load format **L**:



0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15 16 17 18 19 20 21	22	23 24 25
opCode	addrReg	offset	c	F

- Store format **S**:



0 1 2 3 4 5	6 7 8 9 10	11 12 13 14 15	16 17 18 19 20	21	22	23 24 25
opCode	addrReg	valReg	offset	`0`	c	F
opCode	addrReg	imm	offset	`1`	c	F

c . . . condition: `1` . . . if `true`, `0` . . . if `false`
F . . . condition flag to use

2 Register File

2.1 General-Purpose Registers

Index	Name	Purpose
0	r0	global reg 0
1	r1	global reg 1
2	r2	global reg 2
3	r3	global reg 3
4	r4	global reg 4
5	r5	global reg 5
6	r6	global reg 6
7	r7	global reg 7
8	r8	global reg 8
9	r9	global reg 9
10	r10	global reg 10
11	r11	global reg 11
12	r12	global reg 12
13	r13	global reg 13
14	r14	global reg 14, frame pointer
15	r15	global reg 15, stack pointer
16	r16	local reg 0
17	r17	local reg 1
18	r18	local reg 2
19	r19	local reg 3
20	r20	local reg 4
21	r21	local reg 5
22	r22	local reg 6
23	r23	local reg 7
24	r24	local reg 8
25	r25	local reg 9
26	r26	local reg 10
27	r27	local reg 11
28	r28	local reg 12
29	r29	local reg 13
30	r30	local reg 14
31	r31	local reg 15, reserved

2.2 Special Registers

Index	Name	Purpose
0	$c0	Condition flag 0, global, always true
1	$c1	Condition flag 1, global
2	$c2	Condition flag 2, global
3	$c3	Condition flag 3, global
4	$mem	Memory load result as int32_t, read only, global
5	$memhu	Memory load result as 2 × uint16_t, read only, global
6	$memhs	Memory load result as 2 × int16_t, read only, global
7	$membu	Memory load result as 4 × uint8_t, read only, global
8	$membs	Memory load result as 4 × int8_t, read only, global
9	$mul	Multiplication result, per-cluster
10	$rb	Return base, global
11	$ro	Return offset, global
12	$ba	Base address, read only, global
13	?	
14	?	
15	?	
16	$f0, $d0	FPU register 0
17	$f1	FPU register 1
18	$f2, $d1	FPU register 2
19	$f3	FPU register 3
20	$f4, $d2	FPU register 4
21	$f5	FPU register 5
22	$f6, $d3	FPU register 6
23	$f7	FPU register 7
24	$f8, $d4	FPU register 8
25	$f9	FPU register 9
26	$f10, $d5	FPU register 10
27	$f11	FPU register 11
28	$f12, $d6	FPU register 12
29	$f13	FPU register 13
30	$f14, $d7	FPU register 14
31	$f15	FPU register 15

3 Operations

Opcode	Name	Fmt	Unit	Semantics
				Arithmetic
00 0000	add	B	A	dest = src1 + src2
00 0001	sub	B	A	dest = src1 - src2
00 0010	s2add	B	A	dest = src1 + src2∗4
00 0011	and	B	A	dest = src1 & src2
00 0100	or	B	A	dest = src1 | src2
00 0101	xor	B	A	dest = src1 ^ src2
00 0110	sl	B	A	dest = src1 << src2
00 0111	sr	B	A	dest = src1 >>> src2
00 1000	sra	B	A	dest = src1 >> src2
00 1001	rl	B	A	dest = (src1 << src2)|(src1 >>> (32-src2))
00 1010	mul	B	A	$mul = src1 ∗ src2
00 1011	carr	B	A	dest = ((int64_t)src1+(int64_t)src2) >>> 32
00 1100	borr	B	A	dest = ((int64_t)src1-(int64_t)src2) >>> 32
00 1101	mask	B	A	masking operation (see Section 3.1)
00 1110	?			
00 1111	?			
				Conditions
010 000	cmpeq	B	A	dest = src1 == src2
010 001	cmpne	B	A	dest = src1 != src2
010 010	cmplt	B	A	dest = src1 < src2, signed
010 011	cmple	B	A	dest = src1 <= src2, signed
010 100	cmpult	B	A	dest = src1 < src2, unsigned
010 101	cmpule	B	A	dest = src1 <= src2, unsigned
010 110	btest	B	A	dest = (src1 & (1 << src)) != 0
010 111	comb	C	A	flag combination operation (see Section 3.2)
				Constants
0110 00	ldi	I	A	dest = imm, signed
0110 01	ldiu	I	A	dest = imm, unsigned
0110 10	ldim	I	A	dest |= imm << 11, signed
0110 11	ldih	I	A	dest |= imm << 21
				Flow Control
0111 00	br	J	J	pc = imm ? pc+offset : target
0111 01	call	B	J,M	$rb = $ba, $ro = pc, $ba = src2, pc = 0
0111 10	callg	G	J,M	$rb = $ba, $ro = pc, $ba = addr∗4, pc = 0
0111 11	ret	B	J,M	$ba = $rb, pc = $ro

Memory Accesses				
10 0000	stm.a	S	M	[addr+offset*4] = val, all caches, int32_t
10 0001	stmh.a	S	M	[addr+offset*2] = val, all caches, int16_t
10 0010	stmb.a	S	M	[addr+offset] = val, all caches, int8_t
10 0011	stm.s	S	M	[addr+offset*4] = val, stack cache, int32_t
10 0100	stmh.s	S	M	[addr+offset*2] = val, stack cache, int16_t
10 0101	stmb.s	S	M	[addr+offset] = val, stack cache, int8_t
10 0110	wb.s	L	M	write back data from stack cache
10 0111	ldm.b	L	M	issue $mem = [addr+offset], bypass caches
10 1000	ldm.d	L	M	issue $mem = [addr+offset], direct mapped cache
10 1001	ldm.f	L	M	issue $mem = [addr+offset], fully assoc. cache
10 1010	ldm.s	L	M	issue $mem = [addr+offset], stack cache
10 1011	ldmg.d	G	M	issue $mem = [addr*4], direct mapped cache
Special Registers				
1011 00	ldx	B	A	dest = src1, src1 refers to special register
1011 01	stx	B	A	dest = src1, dest refers to special register
1011 10	fop	F	F	floating-point operation (see Section 3.3)
1011 11	?			
Global Address Constants				
110	ldga	H	A	dest = address*4, unsigned
—				
111 000	?			
111 001	?			
111 010	?			
111 011	?			
111 100	?			
111 101	?			
111 110	?			
111 111	?			

3.1 Masking Operation

Src2	Semantics
Extract Unsigned Sub-Words	
000 00	`dest = (uint8_t)src1[ 7... 0]`
000 01	`dest = (uint8_t)src1[15... 8]`
000 10	`dest = (uint8_t)src1[23...16]`
000 11	`dest = (uint8_t)src1[31...24]`
001 0-	`dest = (uint16_t)src1[15... 0]`
001 1-	`dest = (uint16_t)src1[31...16]`
Extract Signed Sub-Words	
010 00	`dest = (int8_t)src1[ 7... 0]`
010 01	`dest = (int8_t)src1[15... 8]`
010 10	`dest = (int8_t)src1[23...16]`
010 11	`dest = (int8_t)src1[31...24]`
011 0-	`dest = (int16_t)src1[15... 0]`
011 1-	`dest = (int16_t)src1[31...16]`
Shift Sub-Words	
100 00	`dest = 0; dest[ 7... 0] = src1[7...0]`
100 01	`dest = 0; dest[15... 8] = src1[7...0]`
100 10	`dest = 0; dest[23...16] = src1[7...0]`
100 11	`dest = 0; dest[31...24] = src1[7...0]`
101 0-	`dest = 0; dest[15... 0] = src1[15...0]`
101 1-	`dest = 0; dest[31...16] = src1[15...0]`
Clear Sub-Words	
110 00	`dest[ 7... 0] = 0`
110 01	`dest[15... 8] = 0`
110 10	`dest[23...16] = 0`
110 11	`dest[31...24] = 0`
111 0-	`dest[15... 0] = 0`
111 1-	`dest[31...16] = 0`

TODO Rework these. Only extraction of sub-words is used really.

3.2 Flag Combination Operation

Op	Name	Semantics
00	and	`d = (i1 ^ s1) & (i2 ^ s2)`
01	or	`d = (i1 ^ s1) | (i2 ^ s2)`
10	xor	`d = (i1 ^ s1) ^ (i2 ^ s2)`
11	?	

3.3 Floating-Point Operations

Op	Src2	Name	Semantics
0000	-	fadd	dest = src1 + src2, single
0001	-	fsub	dest = src1 - src2, single
0010	-	fmul	dest = src1 * src2, single
0011	-	fmac	dest += src1 * src2, single
0100	-	dadd	dest = src1 + src2, double
0101	-	dsub	dest = src1 - src2, double
0110	-	dmul	dest = src1 * src2, double
0111	-	dmac	dest += src1 * src2, double
1000	-	fcmp	comparison, single → int32_t (see Section 3.3.1)
1001	-	dcmp	comparison, double → int32_t (see Section 3.3.1)
1010	-	?	
1011	-	?	
1100	-	?	
1101	-	?	
1110	-	?	
1111	0000	fmov	dest = src1, single
1111	0001	fneg	dest = -src1, single
1111	0010	fabs	dest = abs(src1), single
1111	0011	fzero	dest = 0.0, single
1111	0100	dmov	dest = src1, double
1111	0101	dneg	dest = -src1, double
1111	0110	dabs	dest = abs(src1), double
1111	0111	dzero	dest = 0.0, double
1111	1000	rnd	dest = (float)src1, double → single
1111	1001	ext	dest = (double)src1, single → double
1111	1010	si2sf	dest = (float)src1, int32_t → single
1111	1011	si2df	dest = (double)src1, int32_t → double
1111	1100	sf2si	dest = (int)src1, single → int32_t
1111	1101	df2si	dest = (int)src1, double → int32_t

3.3.1 Floating-Point Comparison

Result Bit	Semantics
0	src1 == src2 && !unord(src1, src2)
1	src1 != src2 && !unord(src1, src2)
2	src1 < src2 && !unord(src1, src2)
3	src1 <= src2 && !unord(src1, src2)
4	src1 > src2 && !unord(src1, src2)
5	src1 >= src2 && !unord(src1, src2)
6	!unord(src1, src2)
7	unord(src1, src2)
8	src1 == src2 || unord(src1, src2)
9	src1 != src2 || unord(src1, src2)
10	src1 < src2 || unord(src1, src2)
11	src1 <= src2 || unord(src1, src2)
12	src1 > src2 || unord(src1, src2)
13	src1 >= src2 || unord(src1, src2)

4 Notes

- Stores to the stack are write-back, stores to other caches are write-through. Stores do not pull data into the caches.
- Support for floating-point operations is optional.