Ida Manual

Introduction

Ida is a 32-bit instruction set using a 24-bit address space. Ida Features a comparison slot in all instructions, and a field used either for registers or immediate values. For example, here is a logical left shift of `%t3` by 8 bits into `%t1`, only executing if the two argument registers are equal.

```
CMPS %a0 %a1
SLL ?EQ %t1 %t3 8
```

Exactly how and why this works is explained in this manual.

Bit Widths & Memory

Memory is divided into separate instruction and data memory segments, each of which has its own 24-bit address space. Instruction memory is aligned to 32-bit values, whereas data memory is aligned to 24-bit values. That is to say that instruction memory features 32-bit words and data memory features 24-bit words. In both cases, memory is addressed by its respective word size (as opposed to single byte addressing). This means that each 24-bit memory address corresponds to a specific 24-bit data word in data memory or to a specific 32-bit instruction word in instruction memory. Each general register can contain a 24-bit. Only data memory can be loaded from or stored to.

Here is an example diagram of instruction memory:

ADDRESS	INSTRUCTION
0x000000:	0x6f300000
0x000001:	0x6f4003e8
...	...
0xfffffe:	0xef000004
0xffffff:	0xff000005

And here is data memory:

ADDRESS	DATA
0x000000:	0x5071fc
0x000001:	0x551492
...	...
0xfffffe:	0x786a9e
0xffffff:	0x67f952

Binary Encodings

There are 3 primary instruction types. The type of an instruction is simply defined as the number of arguments it takes.

All instructions are additionally encoded to take either a register or an immediate value for the argument `%RI`. Which encoding is used is given by the 24th bit, the IMM field. When IMM is unset with a value of 0, only the last 4 bits of the instruction are used for `%RI`, to designate a register.

BITS	4	3	1	4	4	12	4
A3-TYPE	OP	?CQ	IMM=0	%RD	%RS	-	%RI
A2-TYPE	OP	?CQ	IMM=0	%RD	-		%RI
A1-TYPE	OP	?CQ	IMM=0	-			%RI

When IMM is set to 1, the immediate value is instead used, and always sign extended wherever possible into a 24-bit value.

BITS	4	3	1	4	4	16
A3-TYPE	OP	?CQ	IMM=1	%RD	%RS	%RI (imm)
A2-TYPE	OP	?CQ	IMM=1	%RD	%RI (imm)	
A1-TYPE	OP	?CQ	IMM=1	%RI (imm)		

The reason for having 3 different encodings is to use as many bits as possible for the immediate fields. In particular, any A1-TYPE instruction can reference an entire 24-bit instruction address or 24-bit data address using just its immediate field.

The purpose of the `?CQ` field in each instruction is explained later.

Instruction Set

Every instruction takes a form such as the following. Some instructions omit %RS, some omit %RD, and any instruction can optionally omit ?CQ.

```
NAME ?CQ %RD %RS %RI
```

The following is the complete instruction set:

OP	INST	TYPE	FUNCTION	NOTE
0	SLL	A3-TYPE	%RD ← %RS << %RI	Shift logical left
1	SLR	A3-TYPE	%RD ← %RS >> %RI	Shift logical right
2	SAR	A3-TYPE	%RD ← %RS >> %RI	Shift arithmetic right
3	RTL	A3-TYPE	%RD ← %RS ↺ %RI	Rotate left
4	RTR	A3-TYPE	%RD ← %RS ↻ %RI	Rotate right
5	AND	A3-TYPE	%RD ← %RS & %RI	And
6	IOR	A3-TYPE	%RD ← %RS | %RI	Inclusive or
7	XOR	A3-TYPE	%RD ← %RS ^ %RI	Exclusive or
8	ADD	A3-TYPE	%RD ← %RS + %RI	Addition
9	SUB	A3-TYPE	%RD ← %RS - %RI	Subtraction
10	LOAD	A3-TYPE	%RD ← MEM(%RS + %RI)	Load word from memory
11	SAVE	A3-TYPE	MEM(%RS + %RI) ← %RD	Store word to memory
12	CMPS	A2-TYPE	%cr ← %RD ? %RI	Signed comparison
13	CMPU	A2-TYPE	%cr ← %RD ? %RI	Unsigned comparison
14	LINK	A1-TYPE	%ra ← %RI	Link to address
15	JUMP	A1-TYPE	%pc ← %RI	Jump to address

Note that shift or rotate amounts greater than or equal to 24 are undefined, due to the fact that registers are constrained to 24-bit values.

After any instruction executes, the program counter advances by 1 (recall that 24-bit instruction memory addresses each refer to a different 32-bit instruction). The only exception to this rule is the JUMP instruction, in which the program counter is not incremented at all but instead is manually set.

It is worth pointing out that LINK can set the entire contents of %ra to any arbitrary value, eliminating the main benefit of including branch instructions. This is particularly useful for loading the address given by a label or for loading a specific 24-bit immediate value.

Instructions have been designed to cover as many operations as possible. For example, computing not:

```
nor $t0 $t1 $0 # MIPS

XOR %t0 %t1 0xffff # Ida
```

Comparison Queries

Every instruction can be conditionally executed depending on the current comparison query by checking the most recent comparison stored in the comparison register `%cr`. If the query is false, the current instruction does not execute. For example:

```
CMPS %zero %t5
ADD ?GT %t2 %t3 %zero
```

This sets `%t2` to `%t3` only if `%t5` is negative. Otherwise, the operation does nothing. Every instruction has a default comparison query of `?OK`, meaning that every instruction by default will execute unconditionally.

NUMBER	NAME	COMPARISON
0	?NO	FALSE
1	?LE	A ≤ B
2	?GT	A > B
3	?NE	A ≠ B
4	?EQ	A = B
5	?GE	A ≥ B
6	?LT	A < B
7	?OK	TRUE

Note that since the bits of `%cr` can never be directly accessed, Ida allows some flexibility in how `%cr` is implemented. When a program begins, before any comparisons have been computed, all comparisons except TRUE are assumed to be false.

Registers

Ida uses 16 general-purpose registers, each containing a 24-bit value.

NUMBER	NAME	USAGE	NOTE
0	%zero	0 Value	Cannot be overwritten
1	%rv	Return Value	
2	%ra	Return Address	
3	%a0	Argument 0	
4	%a1	Argument 1	
5	%a2	Argument 2	
6	%t0	Temporary 0	
7	%t1	Temporary 1	
8	%t2	Temporary 2	
9	%t3	Temporary 3	
10	%t4	Temporary 4	
11	%t5	Temporary 5	
12	%s0	Saved Temporary 0	Must be preserved across calls
13	%s1	Saved Temporary 1	Must be preserved across calls
14	%s2	Saved Temporary 2	Must be preserved across calls
15	%sp	Stack Pointer	

There are also a few special-purpose registers, none of which can be directly accessed.

NAME	USAGE	NOTE
%cr	Comparison Result	Bit-width is implementation-dependent
%pc	Program Counter	24-bit current instruction address

When a program begins, all registers have a value of 0.

Assembly Details

Ida is entirely case-insensitive. Spaces or commas can be used to delimit instruction arguments comments begin with the number sign and continue to the end of their line. Labels directly reference 24-bit instruction addresses. They can be used to refer to any line, and they can be referenced inside of any A1-TYPE instruction. Label references inside instructions must begin with an at symbol. In general, labels can use any character that is not a comment, space, or comma character.

```
FOO:
BAR BAT: SLR %t0 %t0 %t0 # FOO, BAR, and BAT all refer to this addres
         LINK @RET # sets the return address to address RET
         JUMP @BAZ # jump to address BAZ
RET:     ...

BAZ:     ...
```

A few pseudo-instructions are supported, most importantly the jump-and-link equivalent CALL. Note that all of these pseudo-instructions are only circumstantially efficient.

INST	GENERATES	NOTE
HALT ?CQ	R: JUMP ?CQ @R	Halt program (by putting it in an infinite loop)
CALL ?CQ %RI	LINK ?CQ @R JUMP ?CQ %RI R: ...	Jump and link
RTRN ?CQ %RI	IOR ?CQ %rv %zero %RI JUMP ?CQ %ra	Return value
PUSH ?CQ %RD	ADD ?CQ %sp %sp -1 SAVE ?CQ %RD %sp %zero	Push to stack
PEEK ?CQ %RD	LOAD ?CQ %RD %sp %zero	Peek from stack
POP ?CQ %RD	LOAD ?CQ %RD %sp %zero ADD ?CQ %sp %sp 1	Pop from stack
COPY ?CQ %RD %RI	IOR ?CQ %RD %zero %RI	Copy value

The assembler has support for binary (0b...), octal (0c...), hex (0x...), and base 10 constants, both positive and negative. Numeric comparison queries (?07) and registers (%07) are also supported. For all instruction arguments, the assembler will abort if the number is too large to fit in its slot.

The provided assembler jar simply takes in a list of files to assemble and outputs hex files. As an example, this assembles a given Ida files:

```
$ java -jar Ida.jar program.ida
```

The assembler also has a simple emulator that can execute assembled Ida files. It will execute the given assembled program hex file, optionally using the given memory hex file. Note that program hex files can be assembled by the assembler, and memory hex files can be created by editing a memory module in logisim and then exporting it.

The emulator will only exit in one of two conditions: first if execution passes the final instruction in the assembled hex file, and second if a jump-to-itself style loop occurs (such as the result of the HALT pseudo-instruction). Upon exiting, the values of all the registers are printed, and the contents of the data memory are dumped to a file. As an example, this executes the given program:

```
$ java -jar Ida.jar -exec program.hex data.hex
```