Sol Boucher and Evan Klei CSCI-453-01 04/09/14

FabComp: ISA document

1 Overview

Our Fabulous Computation Machine, or FabComp for short, is a 16-bit general-purpose register–based, big-endian CISC architecture. It proudly features an extensive collection of keystroke-saving instructions, 16 general-purpose registers, and 9 memory address modes. Instructions use hybrid encoding, with the opcode word's format fixed, but between 0 and 6 trailing immediate words. Comparisons are done using conditional registers, but we also provide conditional-and-branch instructions for convenience.

2 Instruction format

2.1 Opcode word

Every instruction begins with an opcode word that always follows this format:

opcode word format															
opcode								immediate		am0		am1		am2	
15							8	7	6	5	4	3	2	1	0

meaning of immediate flags	
value	meaning
00	no operands are immediates
01	operand 2 is an immediate value
10	operand 1 is an immediate value
11	operands 1 and 2 are immediates

The immediate flags can override the address modes of operands 1 and 2: if set, the corresponding operand will be an immediate value rather than a register or memory address. Otherwise, the corresponding `am` (address mode) bits are consulted:

meaning of address mode flags		
value	meaning	first immediate instruction word
00	skip: No operand	None
01	immediate address: Memory address	I-type
10	PC-relative: Offset from program counter	I-type
11	other: Immediate word will specify address mode	R-type

2.2 Immediate words

There are three types of immediate words:

I-type immediate word format	
immediate	
15	0

S-type immediate word format			
short immediate 0		short immediate 1	
15	8	7	0

R-type immediate word format			
secondary am	reg0	reg1	reg2
15 12	11 8	7 4	3 0

meaning of secondary address mode flags		
value	meaning	next immediate word(s)
000	Register value: reg0 specifies a register	None
001	Register indirect: Find EA in register reg0	None
010	Scaled: EA calculated as offset from reg0	S-type
011	Doubly scaled: For 2-dimensional arrays with based in register	S-type
100	Auto increment: Find EA in register reg0, then increment	None
101	Auto decrement: Find EA in register reg0, then decrement	None
110	Scaled displacement: EA calculated as offset from immedate	S-type then I-type
111	Doubly scaled displacment: 2-dimensional immediate-based	S-type then I-type

3 Instruction set

We support the following instructions, all of which are encoded with a standard opcode word:

3.1 List of instructions

	opcode	mnemonic	description
0	00000000	HALT	Stop running
1	00000001	AND	Perform bitwise and
	00000010	OR	Perform bitwise or
	00000011	XOR	Perform bitwise xor
	00000100	LSFT	Perform bitwise left shift
	00000101	NAND	Perform bitwise nand
	00000110	NOR	Perform bitwise nor
	00000111	XNOR	Perform bitwise xnor
	00001000	RSFT	Perform bitwise right shift
	00001001	LAND	Performs logical and
	00001010	LOR	Performs logical or
	00001011	LXOR	Performs logical xor
	00001100	RASFT	Perform bitwise right arithmatic shift
	00001101	LNAND	Performs logical nand
	00001110	LNOR	Performs logical nor
	00001111	LXNOR	Performs logical xnor
	00010000	SLT	Compares if one object is less than another
	00010001	SGT	Compares if one object is greater than another
	00010010	SEQ	Compares if two objects are equal
	00010011	SNE	Compares if two objects are not equal
	00010100	SLE	Compares if one object is less than or equal to another
	00010101	SGE	Compares if one object is greater than or equal to another
	00010110	ADD	Perform signed arithmetic addition
	00010111	SUB	Perform signed arithmetic subtraction
2	00011000	BLT	Branches if one object is less than the other
	00011001	BGT	Branches if one object is greater than the other
	00011010	BEQ	Branches if two objects are equal
	00011011	BNE	Branches if two objects aren't equal
	00011100	BLE	Branches if one object is less than or equal to the other
	00011101	BGE	Branches if one object is greater than or equal to the other
3	00011110	PRNT	Prints the value at the specified location
4	00011111	LNOT	Performs a logical not
	00100000	SIZ	Compares if the object is zero
	00100001	SNZ	Compares if the object is not zero
	00100010	NOT	Performs a bitwise not
	00100011	NEG	Performs a arithmetic negation
5	00100100	BIZ	Branch if the object is zero
	00100101	BNZ	Branch if the object is not zero
6	00100110	INCR	Increments the data in the given location
	00100111	DECR	Decriments the data in the given location
7	00101000	JMP	Jumps to the location given
	00101001	JAL	Jumps to the location given and sets the RA
	00101010	CALL	Jumps to the location and puts the address on the stack
8	00101011	RET	Jumps to the location most recently put on the stack
9	00101100	MOV	Moves object from one space to another

3.2 Verification rules

Each numbered grouping in the instruction set table is validated as follows:

0. • No validation
1. • 2-3 ops
2. • 3 ops
   • op0 is not a register
3. • 1 op*
4. • 1-2 ops
5. • 2 ops
   • op0 is not a register
6. • 1 op
7. • 1 op
   • op0 is not a register
8. • 0 ops
9. • 2 ops

4 Memory specifications

The memory space is split into 16 bit-words, arranged big-endian. Each location is identified via a 16-bit address, which points to a whole word. Because addresses are assigned to words instead of bytes, it impossible to have unassigned accesses. The bus to and from memory is 16 bits, and the ISA only supports transferring a single word at a time.

5 List of registers

The architecture exposes the following 16 general-purpose registers:

Register	Description
$a0–$a2	Argument Registers
$v	Return data Register
$s0–$s3	Saved Registers
$t0–$t5	Temporary Registers
$sp	Stack Pointer
$ra	Return Address

*Although the operand is passed in position 0, it must be treated as a source operand

6 Address modes and formats

6.1 Calculating Address Modes

Address Mode	Effective Address
Immediate Address	imm
PC Relative	EA = $pc
Register Indirect	EA = $reg0
Scaled	EA = $reg0 + $reg1 << imm0
Doubly Scaled	EA = Mem[$reg0 + $reg1 << imm0] + $reg2 << imm1
Auto Increment	EA = $reg0 , $reg0 = $reg0 + 1
Auto Decrement	EA = $reg0 , $reg0 = $reg0 – 1
Scaled Displacement	EA = imm + $reg0 << imm0
Doubly Scaled Displacemnt	EA = Mem[imm + $reg0 << imm0] + $reg1 << imm1