The DLX Processor Architecture

Registers and Memory

The DLX architecture provides 32 general-purpose registers of 32 bits each that are named R0 - R31. Two of these registers have special roles:

- The value of register R0 is always zero.
- Branch instructions to subroutines implicitly use register R31 to store the return address.

Memory is divided into words of 32 bits and is byte-addressed, i.e., word addresses are multiples of 4

Runtime Environment

Operating environments often dictate register conventions in addition to those that are hard-coded into a processor architecture. In our case, the following rule applies (in addition to the special handling of R0 and R31):

- Register R30 is set up in advance to point to an area of memory that is usable for global variables. You should allocate your variables at negative offsets relative to this register.
- By convention, register R29 is used as the stack pointer.
- By convention, register R28 is used as the frame pointer.

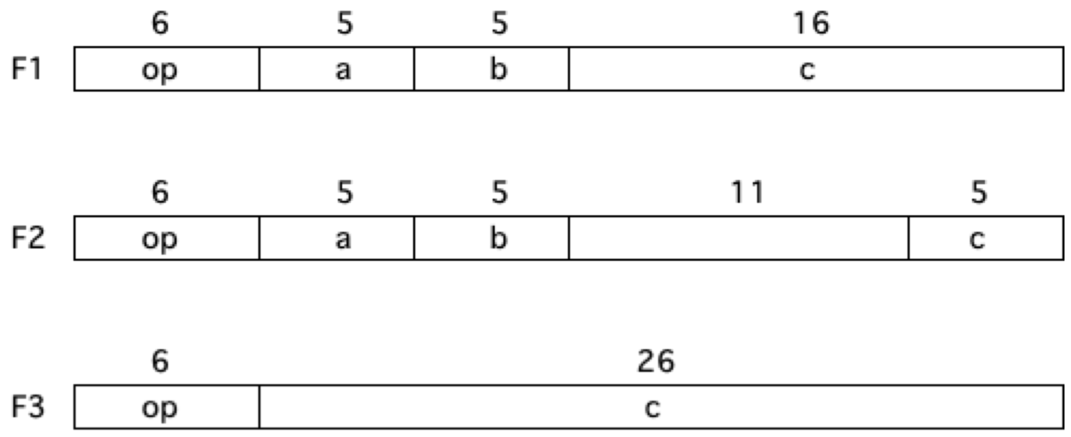
Program Loader

The object file has neither a header nor a trailer. After loading the program, execution starts at the first instruction in the object file (i.e., the one at address zero).

The end of the program is signaled by the instruction RET 0.

Instruction Formats

Instructions are 32 bits in length and start with a 6-bit opcode. The remaining bits depend on the nature of the opcode and follow one of three instruction formats that we call F1, F2, and F3.



Arithmetic Instructions

Arithmetic instructions come in two variants: A variant in which one of the operands is a constant value directly encoded in the opcode ("immediate operand") and a variant in which both operands are contained in registers. The result of the operation is always deposited in a register.

For the two shift instructions, a positive shift count denotes a shift to the left, and a negative count a shift to the right.

Mnemonic		*Operation*	*Fmt/Opcode*	
ADD	a, b, c	R.a := R.b + R.c	F2	0
SUB	a, b, c	R.a := R.b - R.c	F2	1
MUL	a, b, c	R.a := R.b * R.c	F2	2
DIV	a, b, c	R.a := R.b DIV R.c	F2	3
MOD	a, b, c	R.a := R.b MOD R.c	F2	4
CMP	a, b, c	R.a := sign(R.b – R.c) without over/underflow	F2	5
OR	a, b, c	R.a := R.b OR R.c	F2	8
AND	a, b, c	R.a := R.b AND R.c	F2	9
BIC	a, b, c	R.a := R.b AND (NOT R.c)	F2	10
XOR	a, b, c	R.a := R.b XOR R.c	F2	11
LSH	a, b, c	R.a := logical shift of R.b by count R.c	F2	12
ASH	a, b, c	R.a := arithmetic shift of R.b by count R.c	F2	13
CHK	a, c	**halt if** (R.a < 0) or (R.a >= R.c)	F2	14

The CHK instruction is useful for testing the validity of an array index. Execution is aborted if the check fails.

The immediate variants of the instructions use instruction format F1. The 16-bit quantity labelled "c" in the opcode is sign-extended to 32 bits before it is applied to the operation.

Mnemonic		*Operation*	*Fmt/Opcode*	
ADDI	a, b, c	R.a := R.b + c	F1	16
SUBI	a, b, c	R.a := R.b - c	F1	17
MULI	a, b, c	R.a := R.b * c	F1	18
DIVI	a, b, c	R.a := R.b DIV c	F1	19
MODI	a, b, c	R.a := R.b MOD c	F1	20
CMPI	a, b, c	R.a := sign(R.b – c) without over/underflow	F1	21
ORI	a, b, c	R.a := R.b OR c	F1	24
ANDI	a, b, c	R.a := R.b AND c	F1	25
BICI	a, b, c	R.a := R.b AND (NOT c)	F1	26
XORI	a, b, c	R.a := R.b XOR c	F1	27
LSHI	a, b, c	R.a := logical shift of R.b by count c	F1	28
ASHI	a, b, c	R.a := arithmetic shift of R.b by count c	F1	29
CHKI	a, c	**halt if** (R.a < 0) or (R.a >= c)	F1	30

Note that the opcode of an "immediate" instruction can be calculated from the opcode of a "register" instruction by adding 16.

Load/store Instructions

Load/Store instructions use formats F1 for "base register plus displacement" and F2 for "indexed" addressing. In the case of a constant displacement, the 16-bit quantity labelled "c" in the opcode is sign-extended to 32 bits. In the "indexed" case, the contents of a both base registers are added to form the effective address.

Mnemonic		*Operation*	*Fmt/Opcode*	
LDW	a, b, c	R.a := Mem[R.b + c]	F1	32
LDX	a, b, c	R.a := Mem[R.b + R.c]	F2	33
POP	a, b, c	R.a := Mem[R.b]; R.b := R.b + c	F1	34

STW	a, b, c	Mem[R.b + c] := R.a	F1	36
STX	a, b, c	Mem[R.b + R.c] := R.a	F2	37
PSH	a, b, c	R.b := R.b + c; Mem[R.b] := R.a	F1	38

Control instructions

Mnemonic		*Operation*	*Fmt/Opcode*	
BEQ	a, c	branch to c if R.a = 0	F1	40
BNE	a, c	branch to c if R.a <> 0	F1	41
BLT	a, c	branch to c if R.a < 0	F1	42
BGE	a, c	branch to c if R.a >= 0	F1	43
BLE	a, c	branch to c if R.a <= 0	F1	44
BGT	a, c	branch to c if R.a > 0	F1	45
BSR	c	save PC in R31, then **branch** to 4*c (PC-relative)	F1	46
JSR	c	save PC in R31, then **jump** to c (absolute)	F3	48
RET	c	**jump** to address in R.c, end program if c=0	F2	49

Branch means: jump from the current position forward (if c > 0) or backward (if c < 0) by *c* instructions (words). Since instructions can only begin at a word boundary, it makes sense giving the offset in words rather than in Bytes, as this increases the range of the jump by a factor of four.

Jump means: go to an absolute address. Here, the address is given as a Byte address, and not as a word address.

Input/Output Instructions

Mnemonic		*Operation*	*Fmt/Opcode*	
RDD	a	R.a := read a decimal number from the input	F2	50
WRD	b	write the contents of R.b to the output in decimal	F2	51
WRH	b	write the contents of R.b in hexadecimal	F2	52
WRL		start a new line on the output	F1	53