RISC Simulator by Peter Higginson

Instruction Formats (Rd can be Rsd where appropriate)

Hex	Binary Op Code	ASSEMBLY LANGUAGE	DESCRIPTION
00	0000 0	HLT	halt
08	0000 1	MOD Rd, #immediate	modulus
10	0001 0	ADD Rd, #immediate	add
18	0001 1	SUB Rd, #immediate	subtract
20	0010 0	CMP Rb, #immediate	compare
28	0010 1	MOV Rd, #immediate	move
30	0011 0	AND Rd, #immediate	logical and
38	0011 1	ORR Rd, #immediate	logical or
40	0100 0	XOR Rd, #immediate	eXclusive OR (==EOR)
48	0100 1	UDV Rd, #immediate	unsigned divide
50	0101 0	MUL Rd, #immediate	multiply
58	0101 10	LSR Rd, Rs, #count	logical shift right
5C	0101 11	LSL Rd, Rs, #count	logical shift left
60	0110 000	ADD Rd, Rs, Rb	add
62	0110 001	SUB Rd, Rs, Rb	subtract
64	0110 010	AND Rd, Rs, Rb	logical and
66	0110 011	ORR Rd, Rs, Rb (BIS==ORR)	logical or (or bit set)
68	0110 100	XOR Rd, Rs, Rb	eXclusive OR (==EOR)
6A	0110 101	LSR Rd, Rs, Rb	logical shift right
6C	0110 110	LSL Rd, Rs, Rb	logical shift left
6E	0110 1110	ADD SP, #immediate	add
6F	0110 1111	SUB SP, #immediate	subtract
7x	0111	see sub-code list below	
8	100	BRA/B<cond>/JMS, address	branch etc.
A	1010	STR Rs, offset(Rn)	store register
B	1011	LDR Rd, offset(Rn)	load register
C	1100	ADD Rd, direct	add
D	1101	SUB Rd, direct	subtract
E	1110	STR Rs, direct	store register
F	1111	LDR Rd, direct	load register

Branch codes 100x xxx a aaaa aaaa – a is address, x is code (2nd line) – full code in hex 3rd line

BRA	BEQ	BNE	BCS/BHS	BCC/BLO	BMI	BPL	BVS	BVC	BHI	BLS	BGE	BLT	BGT	BLE	JMS
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
80	82	84	86	88	8A	8C	8E	90	92	94	96	98	9A	9C	9E

Branch Meanings

BRA BRanch Always	BEQ Branch if EQual	BNE Branch if Not Equal
BCS Branch if Carry Set	BCC Branch if Carry Clear	BVC Branch if oVerflow Clear
BVS Branch if oVerflow Set	BPL Branch if positive (PLus)	BMI Branch if MInus
BHI Branch if HIgher than	BHS Branch if Higher or Same	BLO Branch if LOwer than
BLS Branch if Lower or Same	BGT Branch if Greater Than	BGE Branch if Greater than or Equal
BLT Branch if Less Than	BLE Branch if Less than or Equal	JMS JuMp to Subroutine

Sub-codes of instruction code 7 (0111)

700	0111 0000 0	ASR Rd, #count	arithmetic shift right
708	0111 0000 1	ROR Rd, #count	rotate right
710	0111 0001 0	INP Rd, address	input
718	0111 0001 1	OUT Rs, address	output
720	0111 0010 000	MOV Rd, flags/SP/LR/PC	move
722	0111 0010 001	MOV flags/SP/LR/PC, Rs	move
7240	0111 0010 0100 0	POP Rd	pop from stack

RISC Simulator by Peter Higginson

7248	0111 0010 0100 1	PSH Rs	push to stack
7250	0111 0010 0101 0	BRA Rs	branch (to address in register)
7258	0111 0010 0101 1	JMS Rs	subroutine call
7260	0111 0010 0110 0000	RET	subroutine return
7261 to 727	0111 0010 0110 0001 to 0111 0010 0111		spare
728	0111 0010 10	MVN Rd, Rs	move NOT
72C to 73	0111 0010 11 to 0111 0011		spare
74x	0111 0100 xx	UDV/MOD/MLX/ASR Rd, Rs	unsigned divide/modulus extended multiply/arithmetic shift right
75x	0111 0101 xx	ROR/DIV/BIC/NEG Rd, Rs	rotate right/signed divide logical bit clear/negate
760/764	0111 0110 0x	INP/OUT Rsd, Ra	input/output
768/76C	0111 0110 1x	CMP/TST Rb, Rs	compare/test (logical and)
77x	0111 0111 xx	MOV/ADC/SBC/MUL Rd, Rs	move/add with carry included subtract with carry included/multiply
78	0111 100	STR Rs, offset(SP)	store register
7A	0111 101	LDR Rd, offset(SP)	load register
7C	0111 110	PSH {R0-R7,LR}	push to stack (bit mask)
7E	0111 111	POP {PC,R7-R0}	pop from stack (bit mask)

Also DAT is an assembler directive to store data and NOP generates MOV R0, R0 (as used by ARM).

Notes

1) The assembler accepts two registers for three register instructions (e.g. ADD Rd,Rs generates ADD Rd,Rd,Rs). Similarly, for example, LSR Rd,#7 generates LSR Rd,Rd,#7. The assembler also accepts (Rn) or [Rn] as equivalent to 0(Rn) and (SP) or [SP] as 0(SP).
2) JMS overwrites the old LR. Standard linkage is to push several registers including LR on entry and pop the registers and PC on exit (so entry LR becomes return PC). So compilers use POP multiple instead of using RET. (Compilers expect the first four parameters go into R0 to R3.)
3) In the inst7 sub-set there are a few spares, the most has 8 parameter bits.
4) While ADD, SUB and MUL are the same for signed and unsigned, DIV is not and so an extra instruction (UDV) is provided for unsigned division. Extended multiply (MLX) is unsigned and clears all flags apart from the Z flag.
5) I used 3 character instructions for convenience. So HALT is HLT and PUSH is PSH.
6) Only ALU operations set the flags. In the ARM implementation of 32 bit instructions an explicit “set flags” bit is required. We are nearer the 16 bit ARM implementation where only some instructions set the flags. LDR and STR do not set the flags (to avoid issues with out of order execution) and MOV does not because the values go nowhere near the ALU. (MVN does set flags.)
7) The instructions ADD/SUB SP,#imm do not change the flags. Execution will error if the SP goes out of the memory range using these instructions. MOV SP,Rx however wraps to stay in memory range as do LDR/STR n(SP) and LDR/STR n(Ra). PSH and POP do not wrap – execution will error (and Reset is then needed).
8) The multi-register PSH and POP instructions take both lists {Ra,Rb,Rc} and/or ranges of registers as parameters but obviously always push and pop in a fixed order. If the PSH was replaced by individual instructions the lowest register would be pushed first (and so end up in the highest address).
9) Output to device 4 is treated as signed but you can output unsigned (device 5), hex (device 6) or character (device 7). Input is a number from device 2.

Instructions in alphabetical order

ADC	ADD	AND	ASR	BCC	BCS	BEQ	BGE	BGT	BHI	BHS	BIC	BIS	BLE
BLO	BLS	BLT	BMI	BNE	BPL	BRA	BVC	BVS	CMP	DAT	DIV	EOR	HLT
INP	JMS	LDR	LSL	LSR	MLX	MOD	MOV	MUL	MVN	NEG	NOP	ORR	OUT
POP	PSH	RET	ROR	SBC	STR	SUB	TST	UDV	XOR				

PLH 12/5/2016