**RISC Virtual Machine**

The RISC machine is deliberately kept simple. There are two classes of instructions, which are separated by bit 31 , the most significant bit. It has a flat memory space of 32 bit words. The first 16 of these are considered to be general registers 32 bit registers R0-R15. There are two flags, Sign and Zero. R15 is the program counter, R14 the link register. This is designed for simplicity not efficiency. It borrows some ideas from my 20 bit RISC for the GI Chip, and some ideas from the ARM chip.

CALL / BRANCH / RETURN

This occurs when bit 31 is ‘1’.

Bits 30,29,28 are the condition bits (see later)

Bit 27 is the link bit. When this is set, R15 is copied to R14 before the branch is performed, effecting an ARM style subroutine call. (return involves copying R14 to R15)

Bits 26…0 are the address to branch to (giving a 2^27 MW or 2^30 byte address space). This address is shifted right twice as all instructions are on 4 byte boundaries.

ALU Instruction

This occurs when bit 30 is ‘0’

Bits 30,29,28 are the condition bits (see later)

Bits 25 and 24 are the function code (see later)

Bits 23..16 modify the source value (in the following order)

* When bit 23 is logic ‘1’ the source byte is sign extended to 32 bits and used as a value.
* Bits 20-16 form a 32 bit value, the source value is rotated left this many times (circularly)

Bits 8..15 specify the target address

Bits 0..7 specify the source address

Addresses

The effective address value is the value of the lower 7 bits if the most significant bit is ‘0’, and the contents of that memory location if the most significant bit is ‘1’ (direct vs indirect). If the indirection is done via R15 it is auto-incremented (allowing MOV R0,@R15 ; WORD 3222)

The exception to this rule occurs when bit 23 of the modifier is ‘1’ then the target then the fetched value is the sign extended value of bits 0..7.

Note all addresses are 32 bit word addresses. 8 bit data will have to be compacted in and out.

Functions

* 00 MOV:Move target to source. No flags change
* 01 ADD:Add source into target. Sets Z and S flags
* 10 AND:And source into target. Sets Z and S flags
* 11 XOR:Xor source into target. Sets Z and S flags.

Condition Codes

* 000 Always
* 001 Z:Execute if zero set
* 010 LT:Execute if sign set
* 011 LE:Execute if zero **OR** sign set.
* 100 Never
* 101 NZ:Execute if zero clear
* 110 GE:Execute if sign clear
* 111 GT:Execute if sign clear **AND** zero clear

Order

All instructions are conditional, so you cannot do a long data move conditionally, because this will mean if the condition fails it will execute the following data byte. Instructions should be processed in this order.

1. Calculate the source address and read it or source value dependent on bit 23
2. Rotate the read value left according to bits 16-20
3. Calculate the target address
4. Calculate the result
5. Set the flags
6. Store in the target

Reset State

The processor is reset to all registers zero, all flags cleared.

Mnemonics

Branch instructions

* B[L][<condition>] address
* <Instruction>[<condition>] [@]target,[@]source or #constant, [ rotate ]

The #constant syntax applies whether it is a short constant in a single word (-128 … 127) or a long constant in the next word. Rotate can be any number, but will be anded with 31 (so you can use -1)

So if you do mov r4,#$2A

It will compile the following which uses the short constant form (broken into bytes for clarity)

$00 $80 $04 $2A

If you do move r4,#$12A it will compile the following words which are move r4,@r15 ; $12A

$00 $00 $04 $8F $00 $00 $01 $2A

Memory Layout

Initial versions of the header data are compiled inline, taking advantage of the fact that any instruction beginning x100 will never be executed.

+0 0100 <shifted right twice address of previous entry. For first entry this is zero.

+4 0100 <first 4 characters of instruction>

+8 0100 <second 4 characters of instruction>

+12 1100 <last 4 characters of instruction>

The words are shifted in so, AT (which is in binary 100 0001 and 101 0011) would be stored as follows:-

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| End+Cond | | | | Unused so all zero. | | | | | | | | | | | | | | A | | | | | | | T | | | | | | |
| C0 | | | | | | | | 00 | | | | | | | | 20 | | | | | | | | D3 | | | | | | | |

Assembler

One element per line. Single pass, reverse patches at end.

[Definition] Compile inline definition for given phrase. Clear all labels.

.nn Label nn where nn = 0 – 99. .nn in a *branch only* mnemonic.

<Mnemonic> Mnemonic code.

+nn Advance code pointer by nnn, padding with 0