LEO-1 Homebrew Computer

Architecture

Rev 1.10, 5th January 2017, John Croudy

The LEO-1 is a complete 16-bit computer system featuring the LEO-1 CPU, a custom-designed CPU. This CPU's architecture was inspired by pioneering RISC CPUs such as MIPS where every instruction is encoded in one word and memory can only be accessed by load and store operations.

The LEO-1 CPU has eight general-purpose 16-bit registers designated R0 to R7 which are visible to the programmer. It also has a 24-bit program counter allowing an addressable memory space of 16,777,216 words. Because all memory operations are performed through 16-bit registers, only 65,536 words can be directly addressed. To allow access to the full 24-bit memory space, two 8-bit Bank registers are provided. The *Instruction Bank* is used when fetching instructions and the *Data Bank* is used when loading or storing data in memory.

There is no hardware stack, but a custom stack can easily be implemented by reserving a register (e.g., R7) for this purpose. Since there is no hardware stack, there are no subroutine call instructions. Indeed, if there were such instructions they would be very difficult to implement since the CPU is not microprogrammed. However, subroutines can be called by copying the PC to registers, adding the necessary offset, pushing the registers onto the custom stack and then jumping or branching. Returning can be done by popping the custom stack into registers, and then jumping through those registers.

Like MIPS, LEO-1 has no condition code register. Conditional branching is done by checking a register and branching on 'zero', 'not zero', 'negative' or 'positive'. Whereas MIPS was designed like this for reasons of optimization, LEO-1 was designed like this for reasons of simplification. [1]

[1] Condition codes complicate the ALU's carry handling. By eliminating them, the ALU doesn't need an external carry-in and the programmer doesn't need to worry about clearing the carry before an add or setting it before a subtract. Since the ALU operation code is limited to 3 bits for register operations, it can't provide both ADD and ADD-with-carry (or SUB and SUB-with-borrow) operations. Therefore SET and CLEAR CARRY instructions would be needed as well. I decided that none of this was worth the extra complexity. Zero and negative flags are not needed if one has a conditional branch instruction that can check a register against zero. The lack of an overflow flag is not of much concern to me. In 35 years of programming I have never typed a single instruction that checked for overflow, but then again, I have never written a compiler or a maths library.

Arithmetic and Logic Unit (ALU)

The ALU has two 16-bit inputs (the ‘A’ and ‘B’ operands), and a 16-bit output. It can perform the following operations depending on a 4-bit code:

Code	Name	Description
`0000`	**B**	Sends the B operand directly to the output.
`0001`	**SUM**	Adds the A and B operands and sends the result to the output.
`0010`	**DIFF**	Subtracts the B operand from the A operand and sends the result to the output.
`0011`	**AND**	Performs logical AND between the A and B operands and sends the result to the output.
`0100`	**OR**	Performs logical OR between the A and B operands and sends the result to the output.
`0101`	**XOR**	Performs logical Exclusive-OR between the A and B operands and sends the result to the output.
`0110`	**LSL**	Performs a logical shift left of the A operand by the number of bits in the B operand (1-8) and sends the result to the output.
`0111`	**ASR**	Performs an arithmetic shift right of the A operand by the number of bits in the B operand (1-8) and sends the result to the output.
`1000`	**B**	Sends the B operand directly to the output (needed for **jump**).
`1001`	**B**	Sends the B operand directly to the output (needed for **banki**).
`1010`	**B**	Sends the B operand directly to the output (needed for **bank**).
`1011`	**SB**	Swaps the high and low bytes of the B operand and sends the result to the output (needed for **swhl**).

Notes

- The ALU does not provide a NOT operation, but this can be achieved by using the XOR operation with $FFFF as the B operand.

- The ALU operations are split into two banks based on the high bit of the operation code. The B operation is repeated in both banks. The main ALU code in an instruction is only 3 bits wide. Register and immediate instructions are limited to using the low bank of operations, 0000 to 0111. Miscellaneous instructions are limited to using the high bank of operations, 1000 to 1111. This is because the top bit of the instruction type field is used as the top bit of the ALU code. This mechanism enables the ALU to be extended beyond its original 8-operation design.

- Shifts are handled in a special way by the ALU. First of all, only the low 3 bits of the operand are used. This allows up to eight different shift amounts. However, a shift of zero actually shifts by eight bits. This keeps the shift amount aligned with the operand value to prevent confusion when the shift amount is in a register (i.e., shift amounts of 1, 2, 3, etc. actually shift by 1, 2, 3 bits.

Instruction format

All instructions are encoded in 16 bits and the top two bits form a code which splits the instruction space into four distinct types, as follows:

`00`	**Register / Memory** Performs ALU operations between registers, placing the result either in a register or in memory. For register instructions, the result is stored in a destination register. For memory instructions, the result is used as the effective address of a register load or store operation.
`01`	**Immediate** Performs ALU operations between a register and an unsigned literal value. The result is stored in a destination register.
`10`	**Jump / Branch** Modifies the Program Counter so that execution jumps to a different address. The program can jump to a 24-bit address specified in two registers or it can branch conditionally or unconditionally.
`11`	**Miscellaneous** These instructions perform special operations such as copying the Program Counter to registers, setting the Bank register, etc.

Type 0; Register / Memory

Assembler	Instruction format	Operation
mov Rc,Rb	**00**ddd000bbb**00**000	Rc = Rb
add Rc[,Ra],Rb	**00**dddaaabbb**00**001	Rc += Rb; Rc = Ra + Rb
sub Rc[,Ra],Rb	**00**dddaaabbb**00**010	Rc -= Rb; Rc = Ra - Rb
and Rc[,Ra],Rb	**00**dddaaabbb**00**011	Rc &= Rb; Rc = Ra & Rb
or Rc[,Ra],Rb	**00**dddaaabbb**00**100	Rc |= Rb; Rc = Ra | Rb
xor Rc[,Ra],Rb	**00**dddaaabbb**00**101	Rc ^= Rb; Rc = Ra ^ Rb
lsl Rc[,Ra],Rb	**00**dddaaabbb**00**110	Rc <<= Rb; Rc = Ra << Rb
asr Rc[,Ra],Rb	**00**dddaaabbb**00**111	Rc >>= Rb; Rc = Ra >> Rb

An ALU operation is performed on one or two registers. The result of the operation is stored in the destination register.

aaa	Register containing operand A.
bbb	Register containing operand B.
ddd	Destination register; accepts the result of the ALU operation.

```
mov r2,r3    ; r2 = r3
add r4,r2,r3 ; r4 = r2 + r3
add r4,r3    ; r4 += r3
sub r4,r2,r3 ; r4 = r2 - r3
sub r5,r6    ; r5 -= r6
```

Assembler	Instruction format	Operation
movpcl Rd	**00**ddd00000**010**000	Move Program Counter to Register

Moves the 16-bit Program Counter to a register before the former is incremented.

ddd	Destination register; accepts the result of the operation.

```
movpcl r0    ; r0 = pc
```

Assembler	Instruction format	Operation
movpch Rd	**00**ddd00000**110**000	Move Instruction Bank to Register

Moves the Instruction Bank to a register. This is essentially the high 8 bits of a 24-bit PC.

ddd	Destination register; accepts the result of the operation.

```
movpch r1    ; r1 = instruction bank
```

Assembler	Instruction format	Operation
mov Rd,(Rb)	**00**ddd000bbb**01**000	Memory read from address in (Rb)
mov Rd,(Ra + Rb)	**00**dddaaabbb**01**001	Memory read from address in (Ra + Rb)
mov Rd,(Ra - Rb)	**00**dddaaabbb**01**010	Memory read from address in (Ra - Rb)

An ALU operation is performed on one or two registers. The result of the operation gives the low 16 bits of an address. The high 8 bits of the address is taken from the Bank register. This forms the 24-bit effective address from which to read data. For more information, see the *Notes* section at the end of this document.

aaa	Register containing operand A.
bbb	Register containing operand B.
ddd	Destination register; accepts the data found at the effective address.

```
mov r4,(r2)    ; r4 = *r2
mov r5,(r2+r3) ; r5 = *(r2+r3)
```

Assembler	Instruction format	Operation
mov (Rb),Rs	**00**sss000bbb**11**000	Memory write to address in (Rb)
mov (Ra + Rb),Rs	**00**sssaaabbb**11**001	Memory write to address in (Ra + Rb)
mov (Ra - Rb),Rs	**00**sssaaabbb**11**010	Memory write to address in (Ra - Rb)

An ALU operation is performed on one or two registers. The result of the operation gives the low 16 bits of an address. The high 8 bits of the address is taken from the Bank register. This forms the 24-bit effective address to which data is written. For more information, see the *Notes* section at the end of this document.

aaa	Register containing operand A.
bbb	Register containing operand B.
sss	Source register whose contents are stored at the effective address.

```
mov (r2),r4    ; *r2 = r4
mov (r5-r6),r0 ; *(r5-r6) = r0
```

Assembler	Instruction format	Operation
nop	**00**0000000000000000	No operation

No operation is performed. This is actually the equivalent of `mov r0,r0` which has no effect.

```
nop
```

Type 1; Immediate

Assembler	Instruction format	Operation
movi Rn,#	**01**rrriiiiiiii000	Rn = #
addi Rn,#	**01**rrriiiiiiii001	Rn = Rn + #
subi Rn,#	**01**rrriiiiiiii010	Rn = Rn - #
andi Rn,#	**01**rrriiiiiiii011	Rn = Rn and #
ori Rn,#	**01**rrriiiiiiii100	Rn = Rn or #
xori Rn,#	**01**rrriiiiiiii101	Rn = Rn xor #
lsli Rn,£	**01**rrr00000sss110	Rn = Rn lsl £
asri Rn,£	**01**rrr00000sss111	Rn = Rn asr £

An ALU operation is performed on a register and an unsigned literal value. The result of the operation is stored in the destination register.

iiiiiiii	8-bit unsigned value (#). This value is promoted to 16-bits without sign extension.
sss	3-bit shift amount (£). This value is interpreted as 1 to 8 (000 means 8).
rrr	Source/destination register; accepts the result of the ALU operation.

```
movi r2,12        ; r2 = 0x000C (from 12 decimal)
movi r3,$C4       ; r3 = 0x00C4 (hex)
movi r4,077       ; r4 = 0x003F (from 077 octal)
movi r1,%10100101 ; r1 = 0x00A5 (from 0000000010100101 binary)
subi r0,6         ; r0 -= 6
addi r3,$99       ; r3 += 0x0099
```

Type 2; Jump / Branch

Assembler	Instruction format	Operation
jump Rb,Ra	**10**000aaabbb00**000**	Jump

Program execution jumps to a new address. The high 8 bits of the address is taken from one register while the low 16 bits is taken from another register.

aaa	Register containing the low 16-bits of the address to jump to.
bbb	Register containing the high 8-bits of the address to jump to (i.e., the bank).

```
jump r1,r0 ; r1=bank, r0=addr. Jump to ((r1 & 0xFF) << 16) | r0
```

Assembler	Instruction format	Name	Operation
bz Rd,label	**10**rrrhhhhhhhh**011**	Branch if zero	Branch if register value is zero
bnz Rd,label	**10**rrrhhhhhhhh**100**	Branch if not zero	Branch if register value is not zero
bpl Rd,label	**10**rrrhhhhhhhh**101**	Branch if plus	Branch if register value is positive (>= zero)
bmi Rd,label	**10**rrrhhhhhhhh**110**	Branch if minus	Branch if register value is negative (< zero)

If a condition is met, program execution branches to a new address in the same bank. If the branch is taken, the branch offset is added to the PC instead of incrementing it. Because the PC is 16 bits, the branch cannot move the PC to a different bank.

rrr	Register containing signed 16-bit value to check.
hhhhhhhh	Branch offset. This is a signed 8-bit value giving a branch range of -128 to +127 words.

```
bnz r1,:loop ; Branch to :loop if r1 is not zero.
bmi r2,:loop ; Branch to :loop if r2 is negative.
```

Name	Assembler	Instruction format	Operation
Branch	**`bra label`**	**`10`**`000hhhhhhhh`**`111`**	Unconditional branch

Program execution branches to a new address in the same bank. The branch offset is added to the PC instead of incrementing it. Because the PC is 16 bits, the branch cannot move the PC to a different bank.

`hhhhhhhh`	Branch offset. This is a signed 8-bit value giving a branch range of -128 to +127 words.

```
bra :loop    ; Branch to :loop.
```

Type 3; Miscellaneous

Assembler	Instruction format	Operation
banki #	**11**000iiiiiiii**001**	Load Bank immediate

The Bank register is set to a specific value. This value will subsequently be used as the upper 8 bits of the effective address of load and store operations.

iiiiiiii	8-bit unsigned value (#)

```
banki $80   ; Bank = $80
```

Assembler	Instruction format	Operation
bank Rb	**11**000000bbb00**010**	Load Bank from register

The Bank register is set from the low 8 bits of a register. This value will subsequently be used as the upper 8 bits of the effective address of load and store operations.

```
bank r0   ; Bank = r0 & 0xFF
```

Assembler	Instruction format	Operation
swhl Rd [,Rb]	**11**ddd000bbb00**011**	Swap register high and low bytes

The ALU operation *SB* is performed on a register. The register's high byte is stored in the low byte of the destination register, and the register's low byte is stored in the high byte of the destination register.

bbb	Register whose high and low bytes are to be swapped.
ddd	Destination register; accepts the result of the ALU operation.

```
movwi r2,$1234 ; r2 = $1234 (movwi is a compound 'asm' instruction)
swhl r2        ; r2 = SB(r2). r2 is now $3412
swhl r5,r2     ; r5 = SB(r2). r5 is now $1234
```

Assembler	Instruction format	Operation
`halt`	**`11`**000000000000**`111`**	Halt the CPU

The CPU halts until it is reset.

```
halt
```

TODO: Display. Large LCD (128 x 64 pixels).

TODO: Keypad.

TODO: Mass storage: IDE hard drive.

TODO: Input and output ports.