Appendix 4
Alecsis assembler

Virtual processor is one of the main parts of Alecsis simulation engine. It emulates the behaviour of a real hardware processor by executing commands successively according to type. The set of legal instructions for the virtual processor comes with adaptations and changes from the set for MC68020, *Motorola* microprocessor.

Text in AleC++ can be translated into the assembly language (assembler) code if you use option '`-S`' (file with the extension '`.as`') in the program call. Besides, you can write assembler code in the text using command **`asm`** (see chapter 2). It is not likely that the user may need assembler commands for modelling and simulation. However**, it may be necessary to understand assembler if the problems arise when installing Alecsis on different computers**.

When using assembler you can encounter some memory problems, problems with honouring various conventions, etc., thus you need to be careful when using it. The use of assembler is tolerated only for writing of very difficult functions whose time of execution is crucial for the program. Compiler sometimes copies interim results into temporary registers to protect them from deleting. This can be more than necessary sometimes, but compiler uses the safer method. Since Alecsis does not have a multipassage optimizer (besides the **peephole optimizer**, which does not deal with the code as a whole but only with 2-3 neighbouring instructions), there is always room for a shorter and more efficient code written in assembler. Optimizing can, however lead you into dangerous waters, which require the knowledge and understanding of the work of virtual processor. Notice that Alecsis always creates the shortest code from the given text (syntax-directed compilation). Mentioned optimizations refer to rearrangement of some expression in order to eliminate extraneous calculations.

A4.1. Operands in assembler instructions

The cycle of the virtual processor divides into two phases: fetching of operands and execution of instructions. Some conventions regulate what can be an assembler operand. Primary (basic) operands refer to a part of memory called *resource* in Alecsis. That is an internal table containing the base addresses off all regions the potential operands can come from. Operation fetch comes down to two additions: one when indexing resource tables; and the other for addition of the address obtained in that way and the operand offset. The position in the table is set using mnemonics pointing the type of the source, while offset is given by a number. Virtual processor supports the following *resource mnemonics* (mnemonic always goes after the character '`%`').

`%d`*n* - general purpose registers (min. 64 bytes)

`%v`*n* - local memory (allocated using instruction `link`)

`%f`*n* - formal parameter (or action parameter, inside a process)

`%m`*n* - object passed to a function or a process (if it fits the declaration)

`%a`*n* - general purpose address registers (min. 64 bytes)

`%s`*n* - local static memory

`%_`*id* - identifier *id* (name with external linking)

`(%a`*n*`)` - content of memory pointed to by the address register (dereferencing)

`%b`*n* - register pointing to ***resource*** vector

`%e`*n* - memory of the *n*-th element (if it has parameters)

`%n`*n* - memory of the value of the signal at the position *n* (in processes) in the current time instant

The last two operands can appear in processes only, since they refer to object that can be declared only in modules. All elements are indexed by the order of declaration, with the index 0 being the first element. Operand points to the pointer to memory of `action` parameters, if the component has them. The last operand points to the memory containing the value of the signal with the index *n* (formal parameters are indexed as 0, 1, 2,.., while the local ones as -1, -2, -3, ...). To get the real address you need to multiply the index and the length of every signal in bytes.

Beside these operands, all expressions of AleC++ can be found in instructions, including constants and variables. For example:

```
int i, j = 2;
asm movq.l i, j;
```

The previous command copies the content of variable `j` into variable `i`. You should avoid complex expressions, since they can use some registers in the way that creates a conflict with the current instruction.

Operand of instructions for conditional or unconditional branching can be name of a label, which can be inside an **`asm`** region or even outside it, but within the original function or a `process`. The instruction itself can have label using the syntax explained in Chapter 3.

The names of functions in the code are created using the mechanism known as **name mangling** (Chapter 4). You can see how this name is created if you compile the file containing the definition of the function using the option '`-S`', and read its name from the assembler file. For example, function:

```
int foo (int, int, double**, class Point, ...);
```

can be called using instruction:

```
asm jsr %dn, %_foo_iiPPd5Point_e ;
```

where `%dn` labels the beginning of memory occupied by the function arguments. The name of the function is prolonged with extensions, that explain which formal parameters are declared for function `foo`. This enables function overloading.

When using temporary registers (`%dn`) you can cross over limit of 64 bytes. Actually, all registers with the index larger than `sizeof(double)` will be transferred to the local memory, allocated as much as needed. This especially refers to passing down the arguments during a function call.

A4.2. Assembler instructions

Most of instructions for virtual processor have more than one version. The current implementation of virtual processor supports the following types: **`b`** (byte), **`l`** (long), **`d`** (double). Some instructions do not have all of these types, or have one as default. The type is appended as the extension of the instruction mnemonics (`add.l`, `move.d`). The type gives information about the number of bytes used by the instruction (sometime the meaning of the instruction, too). The exception is the pair **`mset`**/**`movm`** where you can copy an unlimited number of bytes).

```
mset  64
movm  %_s1,  %_s2
```

In this example, the first 64 bytes pointed to by the external symbol `s2` are copied to external symbol `s1`.

A4.2.1. Instructions of Alecsis virtual processor

mnemonic	syntax	operation	supported types
add	`add.`*t* `op1, op2`	`%d0 = op1 + op2`	`b, l, d`
adda	`adda.`*t* `op1, op2`	`%d0 = op1 += op2`	`b, l, d`
addr	`addr.`*t* `op1, op2`	`%a0 = op1[ op2 ]` Indexing operation has a type, to render multiplication of the index by the type `op1` unnecessary	`b, l, d`
addq	`addq.`*t* `op1, op2`	`op1 += op2`	`l`
asr	`asr.`*t* `op1, op2`	`%d0 = op1 << op2`	`b, l`
asra	`asra.`*t* `op1, op2`	`%d0 = op1 <<= op2`	`b, l`
band	`band.`*t* `op1, op2`	`%d0 = op1 & op2`	`b, l`
banda	`banda.`*t* `op1, op2`	`%d0 = op1 &= op2`	`b, l`
bnand	`bnand.`*t* `op1, op2`	`%d0 = op1 ~& op2`	`b, l`
bnor	`bnor.`*t* `op1, op2`	`%d0 = op1 ~| op2`	`b, l`
bnot	`bnot.`*t* `op`	`%d0 = ~op`	`b, l`
bor	`bor.`*t* `op1, op2`	`%d0 = op1 | op2`	`b, l`
bora	`bora.`*t* `op1, op2`	`%d0 = op1 |= op2`	`b, l`

bxnor	bxnor.t op1, op2	%d0 = op1 ~^ op2	b, l
conb	conb.*t* op	conversion of op from type **byte** into type *t*	b, l, d
cond	cond.*t* op	conversion of *op* from type **double** into type *t*	b, l, d
conl	conl.*t* op	conversion of *op* from type **long** into type *t*	b, l, d
decl	decl.*t* op	%d0 = --op	b, l
decr	decr.*t* op	%d0 = op--	b, l
devb	dev.*t* op1, op2	%d0 = op1 / op2	l, d
deva	deva.*t* op1, op2	%d0 = op1 /= op2	b, l, d
eq	eq.*t* op1, op2	%d0 = op1 == op2	b, l, d
ge	ge.*t* op1, op2	%d0 = op1 >= op2	b, l, d
gt	gt.*t* op1, op2	%d0 = op1 > op2	b, l, d
incl	incl.*t* op	%d0 = ++op	b, l
incr	incr.*t* op	%d0 = op++	b, l
jfn	jfn op1, findx	jump to the intrinsic function with the index *findx* and arguments beginning from the address op1	/
jnz	jnz label	jump to label if %d0 != 0	/
jp	jp label	jump to label	/
jsr	jsr op1, op2	jump to function with address op2 and arguments beginning from address op1	/
jz	jz label	jump to label if %d0 == 0	/
le	le.*t* op1, op2	%d0 = op1 <= op2	b, l, d
lea	lea op1, op2	op1 = &op2	/
link	link.*t* %b0, size	shift of the stack for *t**size bytes	b, l, d
lsl	lsl.*t* op1, op2	%d0 = op1 << op2	b, l
lsla	lsla.*t* op1, op2	%d0 = op1 <<= op2	b, l
lt	lt.*t* op1, op2	%d0 = op1 < op2	b, l, d
mod	mod.*t* op1, op2	%d0 = op1 % op2	b, l
moda	moda.*t* op1, op2	%d0 = op1 %= op2	b, l
move	move.*t* op1, op2	%d0 = op1 = op2	b, l, d
movm	movem op1, op2	copying of content of op2 to op1 - number of copied bytes is determined by the instruction **mset**	/
movq	movq.*t* op1, op2	op1 = op2	b, l, d
mset	mset op	control of instruction **movm**	/
mul	mul.*t* op1, op2	%d0 = op1 * op2	b, l, d
mula	mula.*t* op1, op2	%d0 = op1 *= op2	b, l, d
neg	neg.*t* op	%d0 = - op	b, l, d
neq	neq.*t* op1, op2	%d0 = op1 != op2	b, l, d
not	not.*t* op	%d0 = !op	b, l, d
rts	rts	exit from a procedure	/
sub	sub.*t* op1, op2	%d0 = op1 - op2	b, l, d
suba	suba.*t* op1, op2	%d0 = op1 -= op2	b, l, d
subq	subq.l op1, op2	op1 -= op2	l
unlk	unlk %b0	return of the local memory stack during the exit from a procedure	/
xor	xor.t op1, op2	%d0 = op1 ^ op2	b, l

xora	xora.t op1, op2	%d0 = op1 ^= op2	b, l

A4.2.2. Instructions of Alecsis virtual coprocessor

In the previous section, basic instruction set of virtual processor is given. Beside those instruction, virtual processor has something you can call "coprocessor". Those are additional instructions supporting some of most frequently used functions. This makes a program more effective, especially in the case of mathematical functions, which are used often in modelling of analogue circuits.

mnemonic	syntax	operation	supported types
putchar	putchar.l op	%d0=putchar (op)	l
fputc	fputc.l op1, op2	%d0 = fputc (op1, op2)	l
getchar	getchar.l	%d0 = getchar()	l
fgetc	fgetc.l op	%d0 = fgetc (op)	l
strcpy	strcpy.l op1, op2	%d0 = strcpy(op1, op2)	l
strcmp	strcmp.l op1, op2	%d0 = strcmp(op1, op2)	l
strlen	strlen.l op	%d0 = strlen(op)	l
malloc	malloc.l op	%d0 = malloc (op)	l
calloc	calloc.l op1, op2	%d0 = calloc (op1, op2)	l
free	free op	free (op)	/
attr	attr.l indx, offset	returns the address of user-defined attributes for signals with the position indx and offset offset	l
slen	slen.l indx, offset	returns the length of the signal-vector with the position indx and offset offset	l
fabs	fabs.d op	%d0 = fabs (op)	l, d
exp	exp.d op	%d0 = exp (op)	d
log	log.d op	%d0 = log (op)	d
log10	log10.d op	%d0 = log10 (op)	d
pow	pow.d op	%d0 = pow (op)	d
sqrt	sqrt.d op	%d0 = sqrt (op)	d
sin	sin.d op	%d0 = sin (op)	d
cos	cos.d op	%d0 = cos (op)	d
tan	tan.d op	%d0 = tan (op)	d
asin	asin.d op	%d0 = asin (op)	d
acos	acos.d op	%d0 = acos (op)	d
atan	atan.d op	%d0 = atan (op)	d
atan2	atan2.d op1, op2	%d0 = atan2 (op1, op2)	d
sinh	sinh.d op	%d0 = sinh (op)	d
cosh	cosh.d op	%d0 = cosh (op)	d
tanh	tanh.d op	%d0 = tanh (op)	d
floor	floor.d op	%d0 = floor (op)	d
ceil	ceil.d op	%d0 = ceil (op)	d

You can use coprocessor instructions by honouring standard conventions.

A4.3. Conventions on passing parameters to functions

The mechanism of function call and the return from functions, used by the virtual processor, will be explained on the following example.

```
int z;

main () {
   int x, y;
   z = test (x, y);
}

test (int i, int j) {
   int k;
   k = i + j;
   return (k);
}
```

This code would be compiled as follows (comments in the code are added):

```
_main:
   link.l %b0, 4      // allocating 16 bytes of local space
   movq.l %d8, %v0    // placing variable i (%v0) into the first
       // available register after the accumulator (%d0 to %d7)
   movq.l %d12, %v4   // placing variable j into the next one
   jsr  %d8, _test_ii //call of func. test (the name is completed)
   movq.l %_z, %d0    // result returned via %d0
L0:
   unlk  %b0          // freeing local space
   rts               // end of function

_test_ii:
   link.b %b0, 4     // allocating 4 bytes of local space
   add.l  %f0, %f4   // adding of formal variables i and j
   movq.l %v0, %d0   // storing the result into the variable k
   movq.l %d0, %v0   // return of the result
   jp      L0        // compulsory jump to the output label (to
                     // free the space allocated using link
L0:
   unlk %b0          // freeing local space
   rts              // end of function
```

The arguments are passed in the following manner - they are lined up continually into the register %d*n*, starting from the first free position (the lowest position is `%d8`, since the accumulator occupies the first 8 bytes). Instruction `link` allocates space for all local variables and all interim results that were on locations `%d8`+*n*. In our example function `main` has two local variables of type `int` (2x4 bytes) and uses two positions of a register `%d` to pass arguments (2x4 bytes, totalling 16 - since we used long variant of instruction `link`, this number is divided by `sizeof(long)`). Function `test` allocates space only for its local variable `k`. The result of the function is returned through the accumulator (from `%d0` to `%d7`). After that, the program jumps to label `L0` where it frees local space, and exits from the function. The results larger than 8 bytes return to address `%f0` (address of formal parameters), using instruction `movm`.

The previous example can be realized using combined AleC++/assembler syntax:

```
int z;
```

```
main () {
    int x, y;
    asm {
        movq.l %d8, x
        movq.l %d12, y
        jsr       %d8, %_test_ii
        movq.l z, %d0
    }
}

test (int i, int j) {
    int k;
    asm {
        add.l i, j
        movq.l k, %d0
    }
    return (k);
}
```

This example leaves instructions `link` and `inlk` to the compiler (this is a standard procedure when using **`asm`** command). This applies in both cases to command `return`, too

In pointer arithmetic, you should be careful when dealing with address registers. Register `%a0` is reserved for vector indexing. Therefore, the code:

```
int i,j, a[10], b;
b = a[i+j];
```

compiles to

```
add.l i, j
addr.l a, %d0
movq.l b, (%a0)
```

which means that the instruction `addr` puts the address `&a + sizeof(long)*(i+j)` into the address register `%a0`. The following instruction copies the **content** of that address in the register into variable `b`.

Parentheses can dereference only address registers. To dereference a pointer, you need to transfer it into an address register:

```
int *i, j[10];
j[2] = *i;
```

This code is equivalent to the following code:

```
addr.l j, 2
movq.l %a4, i
movq.l (%a0), (%a4)
```

Note that pointer occupy 4 bytes, so the first free place was `%a4`, after `%a0` had been used.

Instruction `lea` does the **referencing**:

```
int i, j;
j = &i;

lea %d0, i
```

```
movq.l j, %d0
```

Compiler implements referencing in two steps, but it is the standard procedure for the optimizer to merge two steps into one: `lea j, i`.

Built-in functions are called using command `jfn`, and they cannot be mixed with ordinary functions, since their address cannot be obtained. The first operand is the address of the first argument, while the second one is an integer constant used for indexing. Indexes of all intrinsic functions, that are not instructions, are in the file `asm.h` in directory `alecsis/include`. The following is the content of the file.

```
#define _printf         0
#define _fprintf        1
#define _sprintf        2
#define _fflush         3
#define _fopen          4
#define _fclose         5
#define _feof           6
#define _fseek          7
#define _ftell          8
#define _fread          9
#define _fwrite         10
#define _rewind         11
#define _exit           12
#define _system         13
#define _warning        16
#define _drand          17
#define _get_info       21
#define _atof           27
#define _atoi           28
```

The missing indices are used for internal system functions, which only compiler can call. By appending this library you can write:

```
movq.l %d8, "Hello, world!\n"
jfn    %d8, _printf
```

which has the same effect as AleC++ command:

```
printf("Hello, world!\n").
```

You can find all other details linked with using assembler by compiling the source code using option '`-S`' and by direct comparison of source and compiled code. Notice that you cannot use directly the code obtained in this manner. Assembler instructions need to be inside **asm** command for compiler to accept them. The closing example will be a recursive function for calculating the factorial of 170 in double precision:

```
#include <alec.h>

double factor (double i){
    if (i<=1) return 1.0;
    return i * factor (i-1.);
}

int main() {
    double i=170.;
    printf("\tfactor(%g)=%g\n", i, factor (i));
}

//
```

```
// Alecsis assembler code
//
// optimization off
//

// function factor_d

_factor_d:
    link.b   %b0, 8
    le.d     %f0, 1
    movq.l   %d0, %d0
    jz  L1
    movq.d   %d0, 1
    jp  L0
L1:
    sub.d    %f0, 1
    movq.d   %v0, %d0
    jsr      %v0, _factor_d
    mul.d    %f0, %d0
    movq.d   %d0, %d0
    jp  L0
L0:
    unlk     %b0
    rts

//function main

_main:
    link.b   %b0, 28
    movq.d   %v0, 170
    movq.l   %v8, "\tfactor(%g)=%g\n"
    movq.d   %v12, %v0
    movq.d   %v20, %v0
    jsr           %v20, _factor_d
    movq.d   %v20, %d0
    jfn           %v8, 0
L0:
    unlk     %b0
    rts
```