

Computer Architecture Toolkit

Article 2: simpleADL Software Installation

Pete Wilson

Version 1.04 • May 21, 2017

I. Introduction	3
I.1. Folder Hierarchy	3
2. Install simpleADL	4
3. Create a new architecture	9
3.1. Running simpleADL	10
3.2. This Release	13
3.3. Limitations	16

1. Introduction

The software that accompanies these articles assumes a few things:

- You're using a Mac, and you have a sufficiently up-to-date XCode (and its command line tools) installed on that Mac
- You know the basics of using XCode to build 'command-line tools'
- You're comfortable with using the Terminal (no great skill is required)
- A particular folder hierarchy.

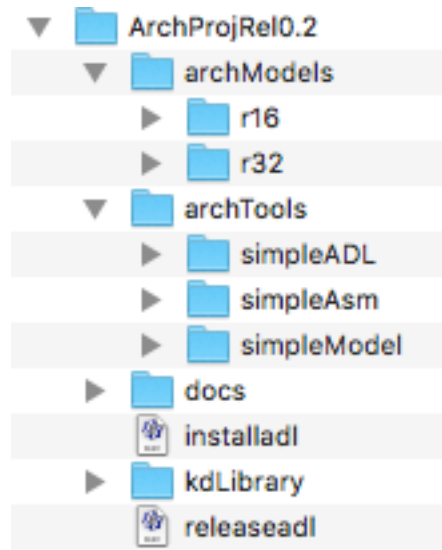
The software is provided as a zip'd file.

- Download it, and then double-click it. It should decompress into a folder hierarchy as described below.
- Move it to where you want.
- Rename the upper level folder if you like (from *ArchProjRelx.x* to whatever you like) (Do NOT change the innards of this folder, though)

WARNING: there must be no spaces in the names of any folder in the folder hierarchy containing *ArchProjRelx.x* and if you change the name, that name must also have no spaces.

1.1. Folder Hierarchy

The folder hierarchy looks like this, where the outer folder may be called something else (and you can change its name):

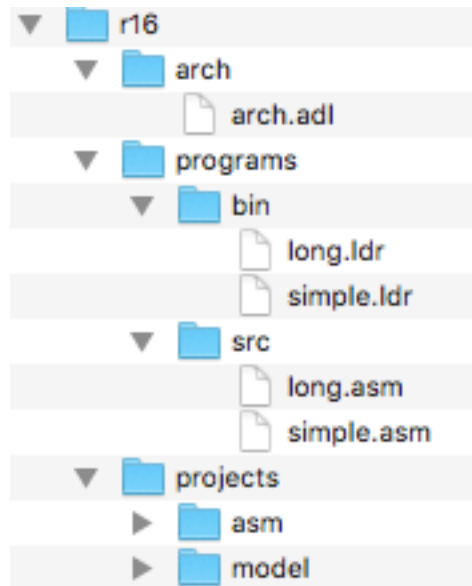


That is, there's a top folder (*ArchProjRel0.2*) which is going to hold all this stuff. It contains four folders, *archModels*, *archTools*, *docs* and *kdLibrary*, along with a couple of files *installadl* and *releaseadl*

- *archModels* contains a folder for each architecture - here, we have just two (*r16* and *r32*). Each architecture will contain its own architecture spec, source code and assembler and executable model. More on this later.
- *archTools* contains a folder for each tool we have - here, *simpleADL*, *simpleAsm* and *simpleModel*. *simpleADL* is the adl compiler. The other two are the canonical assembler and executable model. Each folder contains also a *makefile* and a folder which contains an XCode project for the tool. You can use the XCode project to play with the source code, if you wish.

- *kdlibrary* contains a number of projects which provide common functionality - a queue package, a tokeniser package, a symbol table management package, and a utilities package. These are held in their own XCode projects; when built, these perform simplistic testing of some elements of the packages. The .c and .h files in these projects are incorporated *by being copied* into the simpleADL project, into the simpleAsm, the simpleModel and into the generated projects. This lets you easily see the source of the packages in any project, and it also means that if you make changes to the source of any package, it will NOT be seen by all projects. Be careful!

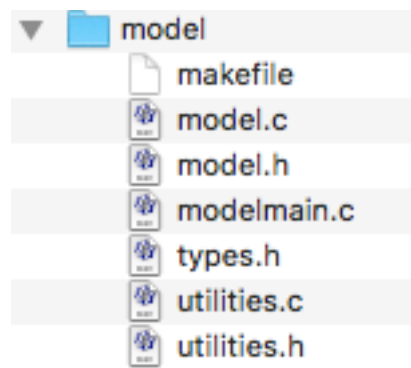
An architecture, such as *r16*, has an internal structure:



Each architecture contains three folders.

- One, *arch*, contains a file *arch.adl* which specifies the architecture.
- *programs* contains a pair of folders, *bin* and *src*; *src* contains the (assembler) source of programs of interest, and *bin* contains their loadable representations.
- The architecture requires an executable model and an assembler; these are collections of source, along with a makefile held in *model* and *asm* respectively inside the *projects* folder.

The *asm* and *model* folders also have a structure. Here's that for *model*:



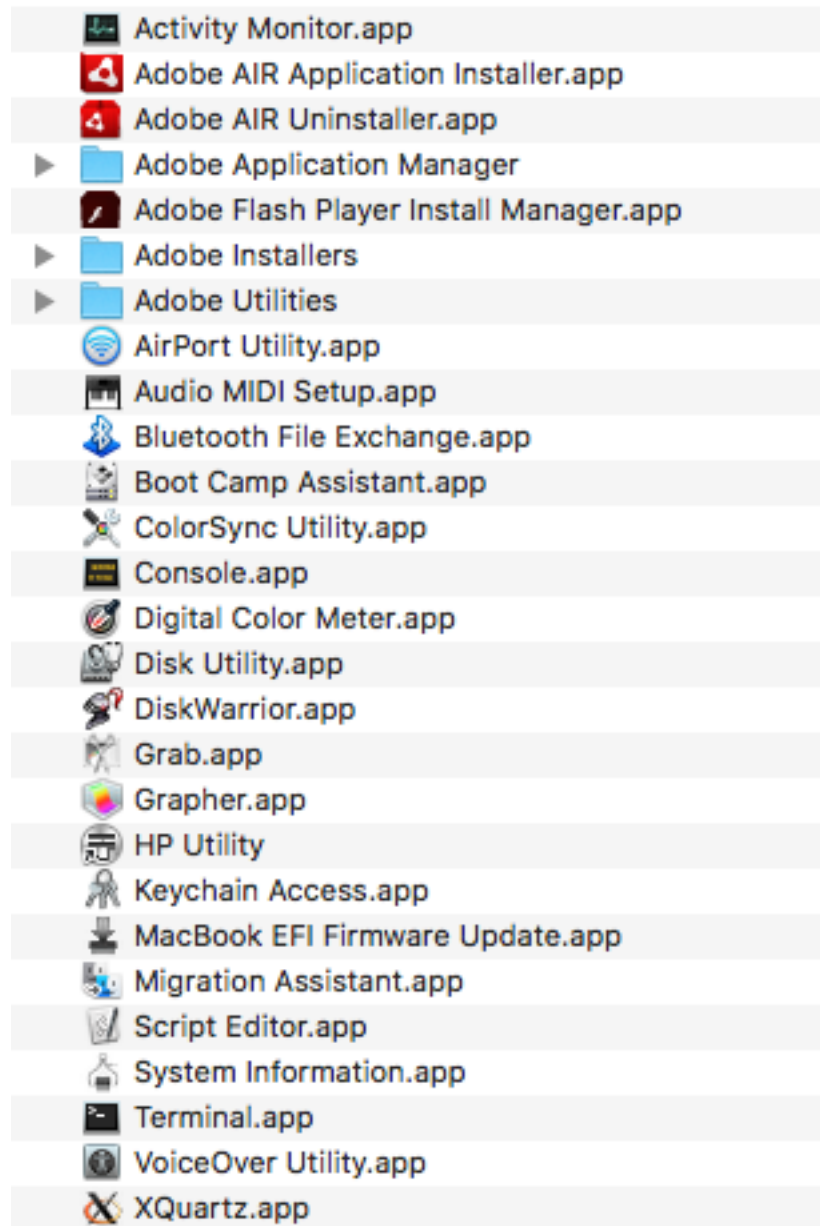
The files *model.c* and *model.h* are created by simpleADL.. *makefile* contains instructions to compile the necessary source files.

2. Install simpleADL

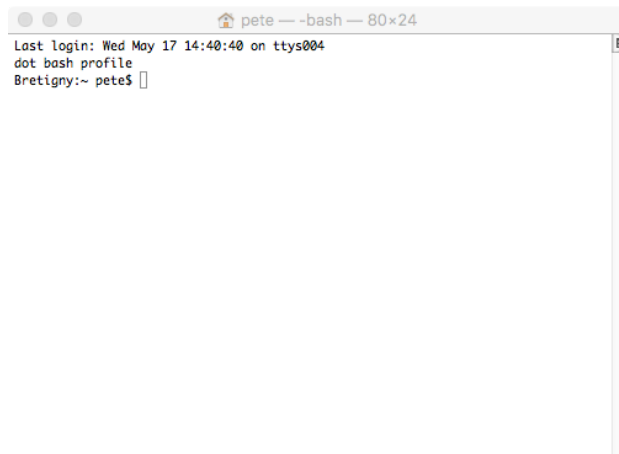
Installing the simpleADL software is pretty straightforward. All you need to do is to open a Terminal window, `cd` to the archProjRel folder you've just unzipped, and type `./installadl`.

If you've not played with the terminal before, it's a program which lets you issue commands to macOS, just like in the good old days of computing before graphical user interfaces.

You will find a folder called *Utilities* inside your *Applications* folder. Open up Utilities and you'll see something like this:



Double-click on *Terminal.app* to run it. You'll get a boring little window something like this:



Click in the window, and type `cd`.

Then open a Finder window and navigate to a view which shows the *ArchProjRel* folder. Drag that folder to the Terminal window, directly after the `cd`. The act of dragging puts the complete path to that folder onto the line of text, thus:

```
Bretigny:~ pete$ cd /Volumes/OxfordRoad/Users/pete/ArchProjRel-0.2
```

Now hit return.

This tells the terminal that you want to operate within the *ArchProjRel* folder.

Type `pwd` and then return. This tells the terminal to **print the current working directory**. On my machine, which is called Bretigny,, we get this

```
Bretigny:~ pete$ cd /Volumes/OxfordRoad/Users/pete/ArchProjRel-0.2
Bretigny:ArchProjRel-0.2 pete$ pwd
/Volumes/OxfordRoad/Users/pete/ArchProjRel-0.2
```

Now type `./installadl` and hit return.

A shell script called *installadl* which is inside the *ArchProjRel* folder will be executed. This tidies up files and copies files into the right places, builds simpleADL and all the other stuff. It'll ask for your password during the installation process, but other than that the whole thing gets done in a few seconds.

The script does install programs in a directory in your machine. You may want to open up *installadl* in a text editor to satisfy yourself it's not doing anything naughty.

You should see something like this happen:

```
Bretigny:ArchProjRel-0.2 pete$ ./installadl
installadl 0.1v0

setting up tools:
copying files for simpleADL...

Making simpleADL...
rm sADL simpleADL.o utilities.o TokUtilities.o TokName.o Tokens.o symbol.o queues.o
generateAsm.o generateModel.o
clang -c -o simpleADL.o simpleADL.c
clang -c -o utilities.o utilities.c
clang -c -o TokUtilities.o TokUtilities.c
clang -c -o TokName.o TokName.c
clang -c -o Tokens.o Tokens.c
clang -c -o symbol.o symbol.c
clang -c -o queues.o queues.c
clang -c -o generateAsm.o generateAsm.c
```

```
clang -c -o generateModel.o generateModel.c
echo "making simpleADL as sADL"
making simpleADL as sADL
clang -o sADL simpleADL.o utilities.o TokUtilities.o TokName.o Tokens.o symbol.o queues.o
generateAsm.o generateModel.o
sudo cp sADL /usr/local/bin
Password:
```

Note that the terminal is asking for your password.

It needs this so the script can copy the simpleADL program into a folder (/usr/local/bin) so it'll be easily accessible from the terminal. The password the terminal wants is the one you use when you start up your Mac. Type it in and hit return. The terminal will continue working. When it's done, you'll have text like this in the terminal:

```
Bretigny:ArchProjRel-0.2 pete$ ./installadl
installadl 0.1v0
```

```
setting up tools:
copying files for simpleADL...
```

```
Making simpleADL...
rm sADL simpleADL.o utilities.o TokUtilities.o TokName.o Tokens.o symbol.o queues.o
generateAsm.o generateModel.o
clang -c -o simpleADL.o simpleADL.c
clang -c -o utilities.o utilities.c
clang -c -o TokUtilities.o TokUtilities.c
clang -c -o TokName.o TokName.c
clang -c -o Tokens.o Tokens.c
clang -c -o symbol.o symbol.c
clang -c -o queues.o queues.c
clang -c -o generateAsm.o generateAsm.c
clang -c -o generateModel.o generateModel.c
echo "making simpleADL as sADL"
making simpleADL as sADL
clang -o sADL simpleADL.o utilities.o TokUtilities.o TokName.o Tokens.o symbol.o queues.o
generateAsm.o generateModel.o
sudo cp sADL /usr/local/bin
Password:
```

```
Copying simpleADL into /usr/local/bin/.. as sADL
copying files for simpleAsm...
```

```
making simpleAsm...
rm sAsm asmmain.o utilities.o TokUtilities.o TokName.o Tokens.o symbol.o queues.o asm.o
clang -c -o asmmain.o asmmain.c
clang -c -o utilities.o utilities.c
clang -c -o TokUtilities.o TokUtilities.c
clang -c -o TokName.o TokName.c
clang -c -o Tokens.o Tokens.c
clang -c -o symbol.o symbol.c
clang -c -o queues.o queues.c
clang -c -o asm.o asm.c
clang -o sAsm asmmain.o utilities.o TokUtilities.o TokName.o Tokens.o symbol.o queues.o asm.o
```

```
Copying simpleAsm into /usr/local/bin/.. as sAsm
copying files for simpleModel...
```

```
Making simpleModel...
```

```
rm sModel modelmain.o model.o utilities.o
clang -c -o modelmain.o modelmain.c
clang -c -o model.o model.c
clang -c -o utilities.o utilities.c
clang -o sModel modelmain.o model.o utilities.o
sudo cp sModel /usr/local/bin
```

Copying simpleModel into /usr/local/bin/.. as sModel

listing architectures:

```
r16...
copying files for asm...
copying files for model...
```

Running simpleADL to create the architecture's asm and model source and header files...

Done. Took 3 milliseconds

Making the assembler and copying to /usr/local/bin

```
rm -f asmr16 *.o
clang -I. -c -o asmmain.o asmmain.c
clang -I. -c -o asm.o asm.c
clang -I. -c -o utilities.o utilities.c
clang -I. -c -o TokUtilities.o TokUtilities.c
clang -I. -c -o TokName.o TokName.c
clang -I. -c -o Tokens.o Tokens.c
clang -I. -c -o symbol.o symbol.c
clang -I. -c -o queues.o queues.c
clang -O2 -o asmr16 asmmain.o asm.o utilities.o TokUtilities.o TokName.o Tokens.o symbol.o
queues.o -I.
sudo cp asmr16 /usr/local/bin
```

Making the model and copying to /usr/local/bin

```
rm -f modelr16 modelmain.o model.o utilities.o
clang -c -o modelmain.o modelmain.c
clang -c -o model.o model.c
clang -c -o utilities.o utilities.c
clang -O2 -o modelr16 modelmain.o model.o utilities.o
sudo cp modelr16 /usr/local/bin
```

```
r32...
copying files for asm...
copying files for model...
```

Running simpleADL to create the architecture's asm and model source and header files...

Done. Took 2 milliseconds

Making the assembler and copying to /usr/local/bin

```
rm -f asmr32 *.o
clang -I. -c -o asmmain.o asmmain.c
clang -I. -c -o asm.o asm.c
clang -I. -c -o utilities.o utilities.c
clang -I. -c -o TokUtilities.o TokUtilities.c
clang -I. -c -o TokName.o TokName.c
```



```
clang -I. -c -o Tokens.o Tokens.c
clang -I. -c -o symbol.o symbol.c
clang -I. -c -o queues.o queues.c
clang -O2 -o asmr32 asmmain.o asm.o utilities.o TokUtilities.o TokName.o Tokens.o symbol.o
queues.o -I.
sudo cp asmr32 /usr/local/bin
```

```
Making the model and copying to /usr/local/bin
rm -f modelr32 modelmain.o model.o utilities.o
clang -c -o modelmain.o modelmain.c
clang -c -o model.o model.c
clang -c -o utilities.o utilities.c
clang -O2 -o modelr32 modelmain.o model.o utilities.o
sudo cp modelr32 /usr/local/bin
```

All done.

Bretigny:ArchProjRel-0.2 pete\$

You should read the output to be sure that there are no complaints. (Complaints like this:

```
rm: sADL: No such file or directory
rm: simpleADL.o: No such file or directory
rm: utilities.o: No such file or directory
rm: TokUtilities.o: No such file or directory
rm: TokName.o: No such file or directory
rm: Tokens.o: No such file or directory
rm: symbol.o: No such file or directory
rm: queues.o: No such file or directory
rm: generateAsm.o: No such file or directory
rm: generateModel.o: No such file or directory
make: *** [clean] Error 1
```

Are not a problem. The remove-a-file command *rm* is moaning that it can't remove a file because it doesn't exist)

When this is done, you can run any of the tools, and the assemblers and models for the architectures, from a terminal window. The assembler for an architecture *X* is called *asmX*; its executable model is *modelX*. All the tools are placed inside the */usr/local/bin* directory, which means you have access to them by typing their names. The programs installed are

- *sADL* - the simpleADL compiler
- *sAsm* - the archetypal assembler
- *sModel* - the archetypal executable model
- *asmr16* - the assembler for the r16 architecture
- *modelr16* - the executable model for the r16 architecture
- *asmr32* - the assembler for the r32 architecture
- *modelr32* - the executable model for the r32 architecture

3. Create a new architecture

To create a new architecture, the simplest thing to do is to

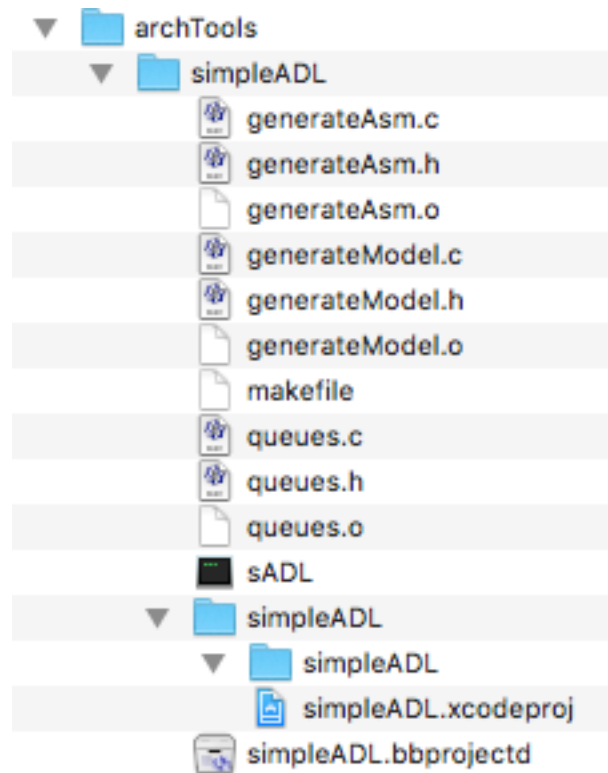
- Duplicate the *r16* folder inside *ArchProjRel*, creating the *r16 copy* folder
- Rename the *r16 copy* folder to the name you want to give the new architecture, say *my_arch*.
- Dive into the *model* and *asm* folders inside the projects folder, and delete the *Derived Data* folder from each if present.

- Edit *arch.adl* in the *arch* folder to reflect the architecture you want
- Run the simpleADL tool, pointing it at your new architecture (by providing the path to the *my_arch* folder). Do this in a Terminal window: simply type *sADL* and then drag the *my_arch* folder into the Terminal window and hit return. Correct the inevitable errors. When eventually executed correctly, simpleADL will write some new files into NewArch's *model* and *asm* folders.
- When it all seems to work, cd back to the ArchProjRel folder and run *./installadl* as you did initially. This will populate your architecture with all the files you need and build the assembler and executable model for it, and install the programs. [It will also do the same for *all* the architectures, but it's quick enough that this does no great harm].
- Modify the example programs that got copied into your asm folder to match your architecture - or write new ones.
- In a Terminal window, type *asmmy_arch* (or whatever) <name of an asm file> to run your generated assembler on the specified file, like *asmmy_arch simple.asm*. Correct the inevitable errors, and then execute the assembled file by typing *modelmy_arch simple.ldr*.

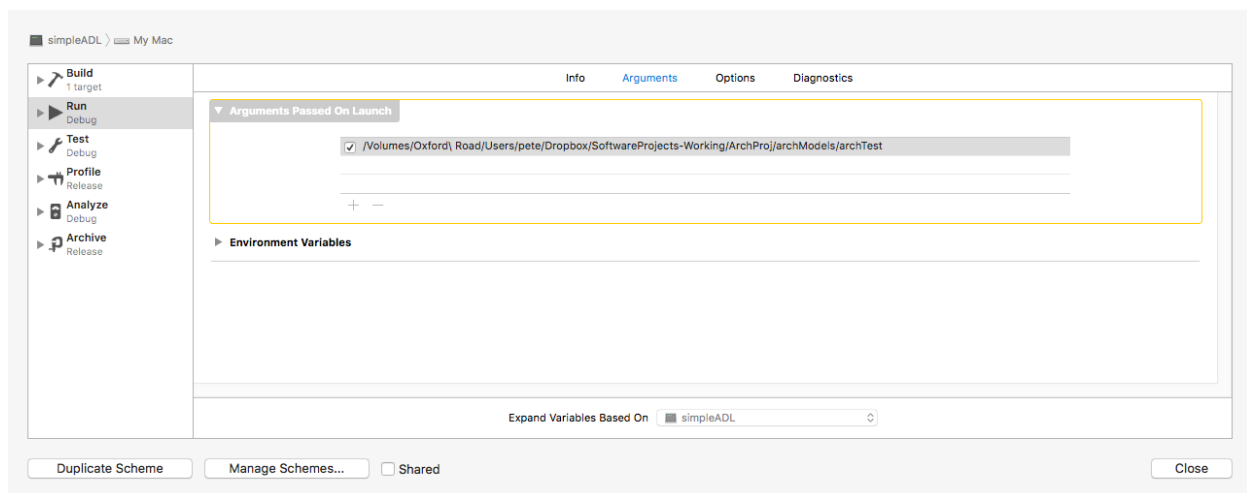
Rinse and repeat.

3.1. Running simpleADL

You can run simpleADL inside XCode. Just double-click on the *simpleADL.xcodeproj* file inside the *simpleADL* folder inside the *simpleADL* folder:



Proceed as usual. You'll need to provide simpleADL with arguments, which you do using the Product:Scheme:Edit Scheme menu and choosing Edit Scheme. Then provide the needed argument(s)



However, the simplest way to proceed is to use a terminal window.

If you type sADL into a terminal window, it will respond with usage instructions:

```
Bretigny:~ pete$ sADL
```

```
simpleADL [simpleADL 1.0v25]
  using utilities kiva utility functions 1.0v4 May 2017
  using tokeniser package simpleTokeniser 1.0v11 January 2017
  using queue package simpleQueues 1.0v1 [May 8 2017]
  using symbol table symbol table management 1.0v1
```

All software copyright Kiva Design Groupe LLC 2017. All rights reserved. See license for terms of use

```
usage:
sADL <options> <path to architecture folder>
options:
  -u -> report release info
  -p -> report parse progress
  -a -> report assembler generation
  -m -> report model generation
  -s -> report statistics
```

Done.

To generate assembler and executable model for an architecture, type sADL into a terminal window and drag the architecture's folder to it, then hit return.

If we do this for the r32 architecture, we get something like this:

```
Bretigny:~ pete$ sADL /Volumes/OxfordRoad/Users/pete/ArchProjRel-0.2/archModels/r32
```

Done. Took 8 milliseconds

```
Bretigny:~ pete$
```

You can select any or all of the options by typing them on the commandline - it doesn't matter what order they're in. You have to specify each one separately. As an example:

```
Bretigny:ArchProjRel-0.2 pete$ sADL /Volumes/OxfordRoad/Users/pete/ArchProjRel-0.2/
archModels/r32 -u -s
```

```
  -u report release info.
  -s report statistics.
simpleADL [simpleADL 1.0v25]
```

using utilities kiva utility functions 1.0v4 May 2017
using tokeniser package simpleTokeniser 1.0v11 January 2017
using queue package simpleQueues 1.0v1 [May 8 2017]
using symbol table symbol table management 1.0v1

All software copyright Kiva Design Groupe LLC 2017. All rights reserved. See license for terms of use

Number of instructions declared = 23
Number of fields declared = 9
Maximum number of fields per instruction = 5
Symbol Table 'architecture' stats:
Number of symbol queues: 29
Total number of symbols: 35
Average symbols per queue: 1
q0: 0 syms [0.000000x the avg]
q1: 1 syms [1.000000x the avg]
q2: 0 syms [0.000000x the avg]
q3: 2 syms [2.000000x the avg]
q4: 2 syms [2.000000x the avg]
q5: 1 syms [1.000000x the avg]
q6: 1 syms [1.000000x the avg]
q7: 1 syms [1.000000x the avg]
q8: 1 syms [1.000000x the avg]
q9: 1 syms [1.000000x the avg]
q10: 3 syms [3.000000x the avg]
q11: 2 syms [2.000000x the avg]
q12: 1 syms [1.000000x the avg]
q13: 0 syms [0.000000x the avg]
q14: 2 syms [2.000000x the avg]
q15: 0 syms [0.000000x the avg]
q16: 0 syms [0.000000x the avg]
q17: 3 syms [3.000000x the avg]
q18: 1 syms [1.000000x the avg]
q19: 2 syms [2.000000x the avg]
q20: 2 syms [2.000000x the avg]
q21: 0 syms [0.000000x the avg]
q22: 2 syms [2.000000x the avg]
q23: 0 syms [0.000000x the avg]
q24: 2 syms [2.000000x the avg]
q25: 0 syms [0.000000x the avg]
q26: 3 syms [3.000000x the avg]
q27: 1 syms [1.000000x the avg]
q28: 1 syms [1.000000x the avg]

Done. Took 3 milliseconds
Bretigny:ArchProjRel-0.2 pete\$

You can be 'in' any folder to run the tools. When you run sADL, the assembler and model files it creates include information on where the relevant architecture folders are. So you can be anywhere and run *asmr32*, for example - it will open the file you specify from the /src folder in the r32 folder hierarchy. This is good, but limiting for real software development. But no-one develops in assembler, and simpleADL is only intended as an educational and proof of concept toolkit.

You can get usage information by running the tools without any arguments. So, for the r32 architecture, we get

Bretigny:ArchProjRel-0.2 pete\$ *asmr32*

```

asmr32 0.1v17
    using tokeniser simpleTokeniser 1.0v11 January 2017
    using queue package simpleQueues 1.0v1 [May 8 2017]
    using symbol table symbol table management 1.0v1

usage:
    -r      -> report all activities
    -s      -> report symbol table statistics
Bretigny:ArchProjRel-0.2 pete$ modelr32

architecture simulator 0.1v6 for 'r32_model 0.1v0'
default bin path '/Volumes/OxfordRoad/Users/pete/ArchProjRel-0.2/archModels/r32'
    usage:
    -l -> report load progress
    -r -> report execution (shortform)
    -t -> trace execution (longform)
    -s -> single-step execution.
    <name of file to load> in /Volumes/OxfordRoad/Users/pete/ArchProjRel-0.2/archModels/
r32

Done.
Bretigny:ArchProjRel-0.2 pete$

```

3.2. This Release

The simpleADL version 0.1v27 improves capabilities compared to previous releases. In particular, it allows the writing of *data* in a program, not just instructions, and the provision of code and data segments. Character constants are also supported.

This requires minor changes in syntax.

Here's what a data-using program looks like:

```

// example r32 assembler program

/*
versions:
0.1v0
    - initial version with data and absolute fixups and code and data segments
*/

title label
memory 0x1000
start 0x10

codeseg

[main]
    cpyc r1, 0;                // allows for 12 bit constant
    cpyc r2, 1000;            // allows for 12 bit constant

[loop]
    addc r1, r1, 1;
    sub r3, r2, r1;
    bne r3, loop;

[finis]
    // some character constants
    cpyc r5, 'a';
    cpyc r5, 'A';
    cpyc r5, '5';
    cpyc r5, '\n';

```

```

    cpyc r5, '\0';
    cpyc r5, @datalabel; // point at the data with r5
    //cpyc r5, 0x7c;
    cpyc r1, 0;
    cpyc r2, 32;
    outcharc '\n';
    outhex r5; // output data start address
    outcharc '\n';
[counting]
    ld8 r3, r5, r1; // read the byte in mem
    outhex r1; // output the address
    outcharc '\t'; // space
    outhex r3; // the value
    outcharc '\n'; // newline
    addc r1, r1, 1;
    sub r4, r2, r1;
    bne r4, counting;

    halt;

```

dataseg

[otherlabel]

[datalabel]

```

d8 1 2 3 4 5 6 7 8 9 10;
d16 0x100 0x200 0x300 0x400 0x500 0x600;
d32 0x12345678 0x4567890 0x98765 0x77 0x1235;

```

[lastlabel]

Changes are shown in bolded text; in summary:

- Programs must now declare **codeseg** before any instructions or data
- Programs must declare **dataseg** before any data
- Data is declared as a sequence of 1, 2 or 4-byte values by the keywords **d8**, **d16** and **d32**. Values may be integers or hex values. The values are *space-separated* and terminated by a semicolon
- The *absolute address* of a label may be captured into any instruction which loads a constant by using the syntax (e.g.) **cpyc @labelname**. No provision is made for labels whose addresses are larger in size than permissible constants.

The assembler has more options than previously. In particular, it can insert labels into the .ldr file, commented out. With this option, the .ldr file for the above is

```

title label.ldr
arch r32
start 0x10
memory 0x1000
codeseg 0x10 0x74
dataseg 0x94 0xbe
code
// [main] at 0x10
    0x20 0x40 0x60 0x00
    0x20 0x80 0x63 0xe8
// [loop] at 0x18
    0x20 0x42 0x10 0x01
    0x10 0xc4 0x00 0x41
    0x20 0xc1 0x4f 0xf8
// [finis] at 0x24
    0x21 0x40 0x60 0x61

```

```

0x21 0x40 0x60 0x41
0x21 0x40 0x60 0x35
0x21 0x40 0x60 0x0a
0x21 0x40 0x60 0x00
0x21 0x40 0x60 0x94
0x20 0x40 0x60 0x00
0x20 0x80 0x60 0x20
0x20 0x01 0xd0 0x0a
0x11 0x40 0x40 0x20
0x20 0x01 0xd0 0x0a
// [counting]                                at 0x50
0x10 0xca 0x07 0x81
0x10 0x40 0x40 0x20
0x20 0x01 0xd0 0x09
0x10 0xc0 0x40 0x20
0x20 0x01 0xd0 0x0a
0x20 0x42 0x10 0x01
0x11 0x04 0x00 0x41
0x21 0x01 0x4f 0xe4
0x10 0x00 0x00 0x00
end

data                                // start of data segment at 0x94

// [otherlabel]                                at 0x94
// [datalabel]                                at 0x94
0x01 0x02 0x03 0x04
0x05 0x06 0x07 0x08
0x00 0x00 0x09 0x0a
0x01 0x00 0x02 0x00
0x03 0x00 0x04 0x00
0x05 0x00 0x06 0x00
0x12 0x34 0x56 0x78
0x04 0x56 0x78 0x90
0x00 0x09 0x87 0x65
0x00 0x00 0x00 0x77
0x00 0x00 0x12 0x35
// [lastlabel]                                at 0xbe
end
stop

```

And executing it provides the result:

```

Bretigny:ArchProjDev pete$ modelr32 label.ldr

architecture simulator 0.1v6 for 'r32_model 0.1v0'
default bin path '/Volumes/OxfordRoad/Users/pete/ArchProjDev/archModels/r32'
Going to load and execute file '/Volumes/OxfordRoad/Users/pete/ArchProjDev/archModels/r32/
programs/bin/label.ldr'
loading software..loaded.

Run the software ..
0x94
0x0    0x1
0x1    0x2
0x2    0x3
0x3    0x4
0x4    0x5
0x5    0x6
0x6    0x7

```

```

0x7    0x8
0x8    0x0
0x9    0x0
0xa    0x9
0xb    0xa
0xc    0x1
0xd    0x0
0xe    0x2
0xf    0x0
0x10   0x3
0x11   0x0
0x12   0x4
0x13   0x0
0x14   0x5
0x15   0x0
0x16   0x6
0x17   0x0
0x18   0x12
0x19   0x34
0x1a   0x56
0x1b   0x78
0x1c   0x4
0x1d   0x56
0x1e   0x78
0x1f   0x90

```

..executed 3270 instructions in 830 microseconds = 3 MIPS.

Done.

3.3. Limitations

simpleADL is fraught with limitations. We don't propose to fix them, because we intend a more complete toolkit which will be noticeably more capable. But just to reduce frustrations from discovering limitations, here's a list of some key ones:

Issue	Commentary
simpleADL generates instruction fetch-and-decode code rather than using the code specified in the .adl file (that is, it ignores the adl file sections <i>initial</i> , <i>operate</i> and <i>halt</i>)	True, and this means that (for example) you can't specify an architecture with a delay slot; nor can you say whether <i>iptr</i> is incremented before the instruction executes.
Branch offsets are always in byte distances between instructions	True, and this means you can't specify an architecture in which the displacement is a number of words, rather than bytes.
There's always one operation per instruction	True, and this means you can't specify a VLIW.
The loader format is absolute, not relocatable.	Yes.
The executable model is only for a single processor	Yes.

The executable model doesn't give useful performance info, like ipc or MIPS.	True enough. But it's an <i>architecture</i> simulator, not an <i>implementation</i> simulator. You'll need to wait a bit for an implementation simulator, and for the ability to model things like caches. But you can compare architectures. For example, simpleRISC executes way fewer instructions than archTest for that trivial program. Same number of loads and stores, though. And much bigger code footprint for simpleRISC.
There doesn't seem to be a way to write, say, <i>addi r7, 67503</i> in assembler source and have the assembler treat this as a pseudo instruction which might end up as a single op (if the literal field in my architecture's add immediate instruction is large enough) or an instruction sequence to build up the literal value somehow.	Yes. Annoying. Sorry. SimpleADL is seen mostly as a "oooh so that's how it's done" educational toy than a real world tool.
So I can write any C I like in the semantics definition of an instruction?	Yes, and that's another thing that will likely change. Having to correctly parse and translate the complete C language seems like much too much hard work for an architecture specification tool, which is why simpleADL doesn't even bother. But we will likely reduce the scope of what you can write, and how it's written in a more realistic tool, all the while keeping an eye on compiler generation and pipeline descriptions (when we get round to implementation models).
How about interrupts and exceptions?	Sorry, we don't do those in simpleADL.
How about MMUs and caches?	Caches <i>shouldn't</i> generally be part of an architecture because running your code with or without caches should in general only have a performance impact, not a change in the results - at least for uniprocessors (although aspects are architected for multiprocessors because coherence and the like). MMUs should be part of the architecture, but we got lazy. It should be pretty straightforward to add an MMU into your architecture, perhaps by writing a <i>map()</i> function which walks the MMU tables/PTEs and is called by a new version of the <i>readMem()</i> function. But then you'll have to write a lot of code, in assembler, to actually make use of the MMU. And you will need to add in exceptions/interrupts, too.
I don't see a test suite. What makes you think this thing is correct?	It probably isn't, but (excuses, excuses) the bugs shouldn't affect the general structure of the software, so (excuses, excuses) it's still OK as an educational tool. Plus, finding the bugs is good for you.
I see <i>readMem()</i> and <i>writeMem()</i> as operations, but neither architectures have any store operations, and there's no example program which accesses memory. Do these reflect a fundamental issue?	Nope. We just got lazy and wanted to get something posted. This is one limitation that will get fixed in <i>simpleADL</i> , probably using the Sieve of Eratosthenes as the example program.

How about that compiler generator, then?	Not in <i>simpleADL</i> . Guessing at the semantics from the instruction semantics is way too complicated for an educational tool.
--	--