Architectural Modelling using LARD

Phil Endecott, University of Manchester.

`pbe@cs.man.ac.uk`

LARD on the Web:

`http://www.cs.man.ac.uk/amulet/projects/lard`

Aim of this presentation:

- Background to LARD and its role in the AMULET3 project.
- Introduction to the LARD language and toolkit.
- Using LARD to model and evaluate a simple microprocessor.

Architectural Modelling for AMULET1 and AMULET2

AMULET1 and AMULET2 were modelling using a hardware description language called ASim. The systems were described structurally in terms of micropipeline elements.

Advantages:

- It is relatively easy to convert from ASim to schematics.

Disadvantages:

- It takes a long time to go from architectural ideas to working models, which makes experimentation time consuming.

The AMULET3 Model

AMULET3 has been modelled in LARD, using a largely behavioural description style. This model

- consists of about 10 000 lines of LARD;
- has taken an order of magnitude less time to develop than the ASim models;
- runs ARM programs at several instructions per second;
- has allowed us to do more architectural experimentation than in the past.

The LARD Requirements

To get these improvements we needed a language and toolkit that:

- Provides channel communication.
- Has high-level language features such as structured types, hierarchical scope, etc.
- Has productivity features such as a debugger, trace viewer etc.
- Can cope with the complexity of the AMULET3 design.

It also had to:

- Be flexible enough to allow us to change it later when we gain experience.
- Be simple enough to get it up and running quickly.

LARD Language Basics

```
/* comments look like this */
```

- `foo ; sprocket` — sequential composition - do foo and then do sprocket
- `foo | sprocket` — parallel composition - do foo and sprocket concurrently
- `( A ; B ) | ( C ; D )` — use `()` to give expressions a particular precedence
- `n:var(int).` — declare an integer variable called n
- `flag:var(bool).` — other basic types are bool, char, string and void
- `mem:var(int^1024).` — arrays are defined using `^`
- `byte:type=(bit^8).` — type declarations
- `point:type=record(x::int,y::int).` — a record declaration

LARD Basics (2)

- `n:=3;` — assignment uses `:=`
- `n:=16#FFE1;` — hex, octal, decimal and binary constants use syntax base#value
- `mem[0]:=42;` — array indexing uses `[]`, arrays start at 0
- `A:=[0,1,42,55,b+1];` — an array constructor
- `p->x:=0;` — record field selection uses `->`
- `a:=a+(b-c)*d/(e mod f);` — arithmetic operations
- `p:=(q and r) or (s<<2);` — bitwise logical operations

Flow of Control

```
if (a=0) then (
   done:=true
) else (
   a:=a-1
)
```

Conventional if-then-else.

Other relations are `!=`, `<`, `>`, `<=`, `>=`.

```
case n
   when 0 => ( c:=a+b )
   when 1 => ( c:=a-b )
   default ( c:=0 )
```

Case statement.

Labels can be constants or expressions.

If no default is given, an error occurs when no label matches.

```
max:=(if (a>b)then a
   else b)
```

if-then-else and case statements can be used in expressions.

Flow of Control (2)

```
while (a>3 && b<10) do (
   a:=a-1;
   b:=b/2
)
```

while loop.

`&&` is boolean and, `||` is or and `!` is not.

```
forever(
   ....
)
```

Infinite loop.

```
forseq n in 0 to 9 do (
   A[n]:=n*n
)
```

Sequential for loop.

```
forpar 10 do (
   fifostage(forparnum)
)
```

Parallel for loop.

Iteration number is accessed via the forparnum function

Defining Functions

```
init:expr(void)=(
   i:var(int).
   forseq i in 0 to 9 do (
      A[i]:=0
   )
).
```

A function with no parameters and returning no result. i is a local variable.

```
sqr(x:val(int))
   :expr(int)=(
   x*x
).
```

A function taking an integer operand and returning an integer result. x is passed by value.

```
incr(i:var(int))
   :expr(void)=(
   i:=i+1
).
```

A function that increments its integer operand. Note that i is a var parameter.

Defining Functions (2)

```
array_sum(
   A:val(int^(n:val(int)))
   ):expr(int)=(
   i:var(int).
   sum:var(int).
   forseq i in 0 to (n-1) do (
      sum:=sum+A[i]
   );
   sum
).
```

A function that takes an array parameter of any size. The size is known at run-time in the implicit parameter n.

```
duplicate(X:val(T:type))
   :expr(T^2)=(
   [X,X]
)
```

A function that will work for arguments of any type.

Defining Functions (3)

```
do_if_debug(E:expr(void))
   :expr(void)=(
   if (debug_enabled) then (
      E
   )
)
```

This function takes an unevaluated expression as a parameter. This would be called by something like

```
do_if_debug(
   print("Here")
);
```


Channel Communication

LARD's channel communication primitives are similar to Tangram's, with one significant difference. The primitives are:

`C!0`	Send value 0 to channel C. The thread is suspended until the value has been acknowledged by the receiver.
`C?(.....)`	Receive a value from channel C. Wait for a value to arrive and then evaluate the body expression. The sender continues only after the body expression has completed.
`?C`	Return the value currently being received on channel C. This may be used only within the body of the `C?(...)` construct.
`C:var(chan(int)).`	Channels are really variables, so they are declared as such.

Time

Time advances when a thread calls the function `wait_for(delay)`.

The current time is stored in the variable **`now`**.

Describing Hardware

(at last!)

Here is a behavioural model of an incrementer:

```
incr(input:var(chan(int)),output:var(chan(int)))
   :expr(void)=(
   forever(
      input?(
         output!(?input+1)
      )
   )
).
```

Note that this block is not a pipeline stage - there is no latch on the input or the output. You can't describe blocks like this with Tangram's channel model.

Another incrementer

Here's another incrementer:

```
incr(input:var(chan(int)),output:var(chan(int)))
   :expr(void)=(
   n:var(int).
   forever(
      input?(n:=?input);
      wait_for(20);
      output!(n+1)
   )
).
```

This description

- ❑ has a latch on its input.
- ❑ gives an explicit delay of 20 time units.

Concurrency within Behavioural Models

(the lack of this is one of the main problems with VHDL)

```
sort_block(
   in1:var(chan(int)),
   in2:var(chan(int)),
   out1:var(chan(int)),
   out2:var(chan(int))):expr(void)=(
   v1:var(int).
   v2:var(int).
   forever(
      ( in1?(v1:=?in1) | in2?(v2:=?in2) );
      ( out1!min(v1,v2) | out2!max(v1,v2) )
   )
).
```

Arbitration using Select

When the real hardware contains an arbiter we probably have a **`select`** construct in the model:

```
arb_call(
   in1:var(chan(int)),
   in2:var(chan(int)),
   out:var(chan(int))):expr(void)=(
   forever(
      case select(in1,in2)
         when 1 => in1?(out!?in1)
         when 2 => in2?(out!?in2)
   )
).
```


wait_until

wait_until suspends the current thread until an arbitrary condition becomes true.

```
valid:var(bool^8).

fill(slot:var(chan(int))):expr(void)=(
   forever(
      slot?(
         valid[?slot]:=true
) ) ).

empty:expr(void)=(
   slot:var(int).
   forever(
      forseq slot in 0 to 7 do (
         wait_until(valid[slot]);
         valid[slot]:=false;
         writeback(slot)
) ) ).
```


Structural Modelling

A structural model is simply a model made up from instances of other (behavioural or structural) models.

Here is a structural model of a system containing the incrementer from a few slides ago, along with a source and sink (not shown):

```
Cin:var(chan(int)).
Cout:var(chan(int)).

"source" is random_source(Cin) |
"incr" is incr(Cin,Cout) |
"sink" is sink(Cout)
```

- **`is`** is used to define the instance hierarchy. It is optional but helps make the runtime environment more friendly.

The LARD Toolkit

The LARD toolkit comprises four main components:

- A collection of libraries, which implement the functionality described earlier, along with other features such as file i/o.
- A compiler and linker which turn LARD source programs into a bytecode language.
- An interpreter for this bytecode.
- A graphical execution environment, based on the tool control language (tcl), into which the interpreter is embedded.

The Compiler

The compiler works in a similar way to other Unix tools and should be familiar to many users.

`lcd foo.l`	Compile the LARD program foo.l, generating a bytecode file foo.bcode.
`lcd -g -O -lutils foo.l`	Compile the LARD program foo.l, enable source debugging (-g) and optimisation (-O), and link with the library utils.
`lcd -mklib foo.l`	Compile the LARD program foo.l, but make a library for further linking rather than an executable bcode file.

Execution

.bcode files are executed using the program li. There are three modes of execution:

- ❑ Batch mode. The .bcode file is loaded and run until it stops.

```
li foo.bcode run
```

- ❑ Command-line mode. The .bcode file is loaded and a tcl command line is displayed.

```
li foo.bcode
li>
```

- ❑ Graphical mode. The .bcode file is loaded and the tk graphical environment is started. You get a tcl command line as well.

```
li foo.bcode windows
li>
```

The general syntax is

```
li [ bcode file | - ] [ module... ]
```


Features of the Graphical Environment

- ❑ Source level debugging with single-stepping, breakpoints, timestep etc.
- ❑ Channel activity traces
- ❑ Animation viewer
- ❑ Message output from the model
- ❑ Power estimation based on channel activity
- ❑ Easily extended by writing your own tcl code

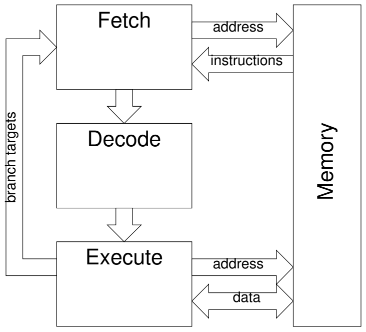
The STUMP Processor

STUMP is a simple 16-bit architecture designed by Doug Edwards for undergraduate lab exercises and other work at Manchester. It features:

- ❑ Fixed length 16 bit instructions.
- ❑ 6 general purpose registers (r1-r6), program counter in r7, and r0 reads as zero.
- ❑ Only explicit load and store instructions access memory.
- ❑ 3-address instruction format, with either **reg = reg op reg** or **reg = reg op immediate**.
- ❑ A shift can be combined with an arithmetic or logical operation.
- ❑ Arithmetic and logical operations optionally set the condition codes.
- ❑ Branches depend on the condition codes and are PC relative.
- ❑ A simple assembler is available.

STUMP Architecture

The stump model presented here has a simple fetch/decode/execute pipeline. This avoids the difficult area of forwarding and register locking, but still leaves plenty of problems for this tutorial!



STUMP Questions

- Does the processor work?
  - Does it deadlock?
  - Does it get the right answer?
- How fast does it go?
  - What limits its performance?
  - How sensitive is the overall performance to changes in particular areas?
- What happens if I change the architecture?
  - What about a single ported memory model?
- Where does the power go?

Conclusions

Much of what LARD does is the tedious stuff that is needed to get a complex model like AMULET3 off the ground. Innovative new approaches to describing asynchronous systems are strictly limited!

Perhaps LARD is only a stopgap before the advent of tools that will automatically synthesize or verify designs, but at the moment we find that manual design is normally the only or most practical approach. Consequently our objective is to provide tools that support human designers. This work has been guided by “what seems useful”.

If you use LARD to model something, I would be very pleased to hear your comments about how useful you found it.