Computer Architecture Toolkit

Article 1: Computer basics and an Architecture Description Language

Pete Wilson

Version 1.01 • May 17, 2017

1. Beginnings - Computers, Architecture and All That

It's fun to argue about computer architecture - or, at least, it used to be. Nowadays, it seems that the architecture debates are settled (you can have an x86, or you can have an ARM. There are those who say you should have a GPU, but they are clearly deluded).

But this is just a temporary stage, similar to when you wanted a mainframe, and your choice was either an IBM 360/370 architecture with an enormous software catalogue, or some weird stuff from a minor company which probably wouldn't stay in business long (like CDC, or Burroughs, or ICL…)

What changed? The simplest explanation is that nobody noticed the macroeconomic effect of Moore's Law. And what happened back then was that minicomputers got as powerful as mainframes, but much cheaper.

Thus the rise of DEC and its competitors.

Then they, too, bit the dust in an horrid fashion as Moore's law continued to operate, and the maligned microprocessor - once a joke to mainframers - grew to match and then outperform minicomputers, again at a much lower price.

So it will be with the mainstream microprocessor architectures, although perhaps for a slightly different reason: Moore still plays a major part - in density - but no longer drives performance. The future will see the death of uniprocessor-tuned architectures (x86, ARM, POWER, MIPS,…) because to make use of the Moore's-provided transistor densities, you need thousands of processors. Which means that architectures which don't provide features which support such numbers of cores will disappear, evolve, or both.

So, if this is correct, we're about to enter a second Golden Age of interesting architectures as folk try to work out what works best in the megacore count era.

Inventing architecture is easy. Understanding the strengths and weaknesses of your own brand new architecture compared with others is much harder, because it requires measurement, which requires program execution, which requires programs and a toolchain and….

So let's get started on a few new architectures…

To do this, we will need to create architectures, and see how well they work, and how they can be improved. To do that properly, we need access to a small but perfectly formed silicon fabrication facility with the ability to build processors (and memories, and I/O hardware, and..) on a number of different silicon processes with feature sizes from multi-micrometer to just a few nanometers at a price suitable for the individual.

Alas, this isn't feasible.

So we'll have to *explore in simulation*.

To exercise our architectures, we'll need to execute programs. To keep the work down, we'll need to have a good-enough compiler for each architecture; to do that, we'll want a compiler-generator - a program which, given a definition of our architecture, generates a version of a compiler which generates code for that architecture.

And to do that, we'll need a formal description of the architecture, so we need an architecture description language (an *ADL*), and another tool which generates a simulator (perhaps several different simulators for looking at different aspects of the architectures) from that architecture description.

This is an approach different from mainstream architecture teachings that I've read. You should immerse yourself in the Patterson and Hennessy books[1] - they're excellent, and they'll look at areas differently from what we do here.

There are also existing architecture description language tools[2], some proprietary. And there's open source professional-quality compilers too[3]. If we were seeking to construct a market-beating architecture, implementation and toolchain, we'd use tools like these. But a professional-quality compiler is large and complex; best to tackle that sort of project once we've done the work of deciding what we want our computer to do in the large. Then we can tune the minutiae and create silicon with the help of substantial funding and much more effort.

So we need tools, and we shall construct them ourselves. This means that they will be relatively simple - and we will choose simplicity over complete generality. It also means we will need a platform on which to develop the tools - and for us, that will be Apple's Xcode suite running on a Mac, with the tools written in C. (The resulting source should 'just work' on a Unix/Linux machine, with the creation of some makefiles.)

Our software tools will share common subsystems. Both the ADL compiler and the program compiler are compilers, and need to scan the source code, comprehend it (while pointing out errors) and build some form of internal representation from which the desired output can be created. We will end up with a number of

---

1. Computer Architecture - A Quantitative Approach-Hennessy and Patterson; and

Computer Organization and Design, Fifth Edition: The Hardware/Software Interface-Patterson and Hennessy

2. NXP's ADL is one open source example: http://opensource. freescale. com/fsl-oss-projects/

3. llvm (http://llvm. org) is the obvious current example, but one must not forget gcc (http://gcc. gnu. org).

subsystems, including a tokeniser (to recognise important source elements), a symbol table (to keep track of all the names in a program and their properties) a tree (a tree is a convenient internal representation of the source of a block-structured program), a linked list (convenient for holding instructions generated in the compiler, making reordering and insertion/removal simple) and more.

To start with, we'll look at the basics of computer architecture, and build a simple ADL language and compiler, which will create an interpreter (so we can execute programs) and an assembler (so we can write programs). We won't push these into full-fledged, reliable tools, discussing and developing them just far enough to make it clear that they could be developed, and to show how we attack the issues of creating them

Later, we'll build significantly more general and robust versions of the tools, and add to them.

2. Computer Basics

A computer is a *machine* which has *instructions* held in a *memory*. The instructions encode *operations* and *resources*.

- Operations are the computational activities, like add, or multiply.
- The resources are the chunks of hardware that hold values to be used in the computations; generally, they are some form of memory.

A memory is a hardware resource which can hold values; a memory can be characterised (at this stage of discussion) as having a capacity, usually measured in bytes, and usually a power of two number of bytes (a byte is just eight binary bits). The bigger a memory is, the slower it tends to be. When your computer needs to have thousands of millions of bytes of capacity, it generally needs to be constructed (for economic reasons) from relatively slow memory hardware[4].

It's perfectly possible to construct a computer whose instructions specify that the computational operation desired, such as add, is to be applied to three elements of memory[5]. But generally, computers provide a much smaller capacity, much faster memory, often referred to as the *general purpose registers*, which are preferentially used as the *operands* of an instruction. In a computer with 32 registers, one might specify instructions which demand a variety of operations on the registers, such as:

- add r1 to r7 and put the result in r25 (where by r1, we mean "register 1")
- multiply r25 by r13 and put the result in r23
- subtract r9 from r11 and put the result in r15

. . and so forth, where when we say 'add r1 to r7', we're writing shorthand for 'add the value in r1to the value in r7'.

The computer also has a resource called the *instruction pointer*[6] (abbreviated here to *iptr*), it contains the address in memory of the instruction which the computer should next execute.

To execute an instruction a computer:

- reads the instruction from the address in the memory specified by iptr
- decodes the instruction - identifies the operation it is to perform and the resources it is to use for the computation
- Reads any resources which supply values
- performs the operation
- Writes the result to the specified resource
- changes the *iptr* appropriately - generally, advancing it to the next instruction

In principle, we can encode the operations we want into instructions any way we wish. We can also choose an instruction to be a fixed size, or a variable size. The bigger the instruction word, the more operations and operands it can specify; the smaller, the less space an average program might need in memory. We'll look at why you might choose which of these schemes later on, but will start - for simplicity - with a discussion of an architecture in which every instruction is encoded into just 16 bits.

Given a 16 bit instruction word, we could choose to specify that of the available values:

- an instruction with value 0 means add r1 to r7 and put the result in r25
- an instruction with value 1 means multiply r25 by r13 and put the result in r23
- an instruction with value 2 means subtract r9 from r11 and put the result in r15

---

4. What counts as 'relatively slow' varies with technology. Back in the 1960's and 1970's, plated wire memories were very very fast at hundreds of nanoseconds access time. . .

5. The Ferranti 1600 series computers were one example of this. We'll see more of them later on.

6. For historical reasons, this is often called the program counter. But of course it doesn't.

. . and so on, adding operations as we discover new ones we want to have performed.

But decoding something (decoding = working out what the instruction is supposed to do) has a cost. We can minimise the cost of decoding instructions like these by recognising that all the instructions have four components - one operation-specifying element, and three register-specifying elements.

If we choose to format our instruction with these four *fields*, we will have significantly simpler hardware[7]. Since we have (in this example) 32 GPRs, we only need five bits to specify any one of them, so 15 bits can define a destination register and the two source registers. Alas, that wouldn't leave many bits to specify the operations, so we'll need to do something different.

One option is to use a 32 bit instruction word, but that's a little expensive in bits. 24 bits is a better fit (it has other problems, but we'll get to those later). This gives us room for encoding 512 different operations, thus:

23	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
operation									rd					rs1					rs2				

Now, whatever hardware we use to select the source registers, read their values, present them to the operations hardware, and then write the result to the specified destination register is the same for every instruction - major saving in wires and transistors. Note that we're numbering the bits in the instruction in a particular order. In our case, we're numbering the bits so that if you read an instruction out of memory and look at it as a number, rather than instruction, the bit numbered 0 has the value 1, the bit numbered 2 has the value 2, and in general bit n has the value 2^n.

But it turns out that we need to do more than compute from values in registers. In particular, we need to

- be able to specify constant values as operands
- be able to read and write memory
- be able to look at a value, and decide to do *this* or to do *that*

We need the constant values for obvious reasons. We can't "add one to r3" if we can't specify that the value to add is 1. You might think that one way of doing this is to put all the magic values we will need into memory; our compiler toolkit would do this for us. Then, to add 1 to r3, we would need to

- read that value from memory and put it into a register, say r1
- perform *r3 = r3 + r1*

There's nothing that says we can't specify an instruction to do this, but before getting overly excited, let's think one step further: where in memory do we have the value '1' stored? And how do we access *this* memory location rather than *that*?

The locations in memory are numbered, conventionally from 0[8], and increasing by one for each individually-accessible element of memory. These days[9], memory is generally accessible byte by byte, and so successive elements are bytes and are numbered 0, 1, 2… Often as not, a memory will support the reading of 1, 2, 4 or 8 elements at once - in our case, chunks of 8, 16, 32 or 64 bits. You tell the memory that you want to read or write 8, 16, 32 or 64 bits starting at byte address *n*, and the hardware obeys your command. If you want to read a 128 bit chunk, you'll need to make multiple accesses; if you want to access a 19 bit chunk, you'll need to read the chunk one size up (a 32 bit chunk, in this case), and use further instructions to strip out the bits that you don't want.

So to read our value 1 from memory, we need to read it from the location it's in. And that is another constant value. Now we're stuck in a never-ending recursive tailspin; obviously, we need some way of putting constants into at least some instructions.

Suppose we're into simplicity; then we could say that we'll be happy with 10-bit constants, and so we could have a different-shaped instruction which is of this form:

23	*22*	*21*	*20*	*19*	*18*	*17*	*16*	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
operation									rd					10-bit constant									

7. Don't believe me? Find a hardware guru to sketch the two solutions; or (better) beg access to a tool which can synthesize hardware from a System C specification; then write a simplified SystemC description of what the computer has to do, and look at the areas of silicon needed with whatever default library the tool comes with.

8. The Inmos transputer had signed addresses; memory ran from -bignum, through 0, to + bignum. It let you use normal signed arithmetic for address arithmetic.

9. Our F1600 accessed memory only as (24-bit) words.

Then some of the operations could mean *rd = r1 + r2* and some could mean *rd = constant*. If we need a value that's bigger than 10 bits, we can load the top 10 bits into (say) r1 and then the lower 10 bits into (say) r2, and then compute the 20 bit value we want by an operation like

```
r1 = (r1 << 10) + r2; // magic shift'n'add instruction
```

Now, to do a memory operation, first we get the memory address (the location of the value we want) into a register, then execute a memory read instruction. Usefully, these are the same shape as the usual add etc instructions - we're just using up more of the operation bits. If we do nothing special, we will need to write a zero value into another register before reading from memory, but we know how to do that:

```
r1 = 317     // the address in memory where we're keeping the value 1
r2 = 0           // we always use two registers to access memory, so we need to          // set up a
second register
r1 = mem[r1 + r2]
r3 = r3 + r1;         // remember? we wanted to add one to r3. . .
```

Now, that's just silly. We can directly include constants into the instruction, so we can actually do:

```
r1 = 1
r3 = r3 + r1
```

There are obvious improvements (for example, the values 0 and 1 are used much more frequently than others…) but let's ignore them for the moment - we still have to manage the *do this* or *do that*.

To do this, we introduce the idea of *branch instructions*. In this example, we'll specify that some of our register-register instructions specify *comparisons* rather than the ordinary arithmetic/logical operations; the *gt* instruction, for example, that has the form:

```
gt r2, r5, r6
```

will write 1 into r2 if r5 > r6, and 0 otherwise. No big instruction design change yet, then.

And now we can specify another instruction which is of the form

```
bt r2, constant      // branch if r2 is 'true'
```

This looks at the value in *r2*, and if it's 0 does nothing; if it's non-zero it *adds* the constant to the current value of *iptr* so that the next instruction to be executed won't be the one immediately after the *bt* instruction.
In general, if we have program which is of the form ***if*** *(something) do <u>this</u>* ***else*** *do <u>that</u>;* then we'll end up with code that looks something like this. It shows the need for another new instruction, *jump,* which always branches.

```
instruction
instruction
compare into r3
bt r3, label3
instruction  // we do these instructions if r3 was zero
instruction
jmp label4   // always branch to label 4
label3:
instruction  // we do these instructions if r3 was non-zero
instruction
instruction
…       // no need for a jump to label4; we just 'fall through'
label4:
instruction  // we want to do these instructions whatever the value of r3
instruction
. . .
```

Because *jump* always branches, it doesn't need to mention a register, and we can use all 15 bits of register specification to specify how far away to jump to:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
operation									15-bit constant														

Looking back at our instructions, we now have three different *instruction formats*. It's probably a good idea to provide a hint to the decoding hardware - we'll reserve the top two bits of the instruction to specify the format, and the others for their current usage:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		operation							rd					rs1					rs2					

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1		operation							rd					10-bit constant									

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2		operation							15-bit constant														

We'll call the 2-bit field of bits 22 and 23 the *format* field. Looking at this, it's pretty clear that if we use format 0 for all the arithmetic, memory access and comparison operations, we need far fewer *operation* bits in the other two formats. In fact, with the instructions we've mentioned so far, we don't need any operation bits in format 2, since we always do a jump. Similarly, as defined, we only ever load a constant into a gpr in format 1, so again we need no operations bits.

But if we make the obvious changes, we will be faced in format 1 with a minor conundrum - the register we're writing to is slap in the middle of the bits specifying the constant. The easiest way of getting round this is to change the register field positions so that the rd-specifying bits are at the bottom….

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		operation							rs1					rs2					rd					

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1		operation							10-bit constant										rd				

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2		operation							15-bit constant														

…and then shrink the operations fields in formats 1 and 2. We'll not actually knock them down to non-existence, because there are certainly some other instructions we'd like to encode:

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0		operation							rs1					rs2					rd					

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1		op		15-bit constant															rd					

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
2		op		20-bit constant																			

We won't worry about the other instructions at the moment; we have enough here for us to think about how we'd want to specify the behaviour of this machine in an architecture description language.

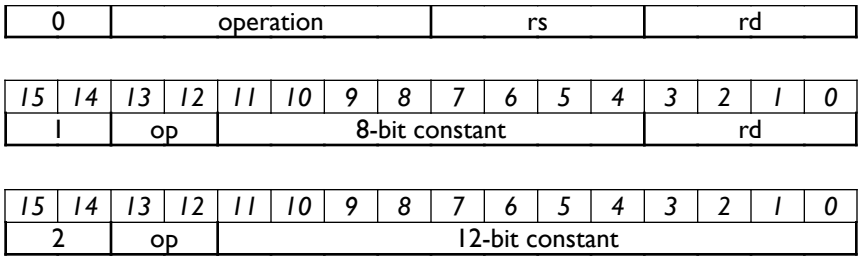
2.1. An Interpreter

An *interpreter* is a piece of program which 'behaves like a processor'.

It reads values from memory, interprets them as instructions (thus the name), and performs the actions called for by the instruction. Naturally, the interpreter doesn't do those actions to the real hardware resources of the computer it's pretending to be - it operates on program variables and data structures which represent the values the real hardware would have.

To make this a little more real, let's consider a simple interpreter. To avoid complications, we'll drop back to a 16 bit instruction word, and just 16 GPRs. And we'll make the arithmetic instructions be of the form *rd = rd <op> rs1*, so we need just 8 bits to represent all the registers in use. We can have an encoding scheme similar to that which we used for our 24 bit example above - 2 bits of *fmt*, followed by 6 bits of *operation* for the register-register instructions, and so forth. So the instructions look like this:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0



The innards of a simple interpreter for this architecture might look like this:

```
typedef struct {
  int32 gpr[16];
  uint32 iptr;
  } Machine;

// ------------------ initModel ---------------

void initModel(Machine * registers) {
  uint32 i;
  for (i = 0; i < 16; i++) registers -> gpr[i] = 0;
  registers -> iptr = 0;
  }

// ---------------------- model --------------------

uint16 model(uint8 * memory, Machine * registers) {
  uint16 instruction;
  uint16 * mem = (uint16 *)memory;
  while (1) {
    instruction = mem[iptr++];
    uint8 format = (instruction >> 14) & 3;
    switch (fmt) {
      case 0:             // arithmetic register-register instructions
        uint8 operation = (instruction >> 8) & 0xff;
        switch (operation) {
          case 0:
            {
            uint8 rd = instruction & 15;
            uint8 rs = (instruction >> 4) & 0xf;
            gpr[rd] = gpr[rd] + gpr[rs];
            }
            break;
          case 1:
            {
            uint8 rd = instruction & 15;
            uint8 rs = (instruction >> 4) & 0xf;
            gpr[rd] = gpr[rd] - gpr[rs];
            }
            break;
          case 2:
            {
            uint8 rd = instruction & 15;
            uint8 rs = (instruction >> 4) & 0xf;
            gpr[rd] = gpr[rd] * gpr[rs];
            }
            break;
          case 3:
            {
            uint8 rd = instruction & 15;
            uint8 rs = (instruction >> 4) & 0xf;
            gpr[rd] = gpr[rd] / gpr[rs];
            }
            break;
```

```
        // should have many more instructions in here

        default:
          return instruction;
          }

    case 1:        // constants and conditional branches
      uint8 op = (instruction >> 11) & 3;
      switch (op) {
        case 0:
          {
          uint8 rd = instruction & 15;
          int8 d = (instruction >> 4) & 0xff;
          if (gpr[rd] != 0) iptr += d;
          }
        case 1:
          {
          uint8 rd = instruction & 15;
          int8 d = (instruction >> 4) & 0xff;
          gpr[rd] = d;
          }
          break;
        default:
          return instruction;
        }

    case 2:        // longer unconditional branches
      uint8 op = (instruction >> 11) & 3;
      switch (op) {
        case 0:
          {
          int16 d = instruction & 0xfff;
          iptr += d;
          }
          break;
        default :
          return instruction;
        }

      default:
        return instruction;
        }
      }
    }
```

Codepiece 1. The innards of a simple interpreter

This is pretty straightforward to understand. We first declare a structure which holds all the state of a processor - in this case, it has 16, 32 bit registers and a 32 bit *iptr*, which holds the address of the next instruction to execute.

We provide an initialisation function, which zeros out all the registers.

And we provide a function *model()*, which interprets memory as a set of instructions. It's implemented as a set of nested *switch* statements. Each level looks at a different field in the instruction, and branches appropriately. If we hit an unimplemented instruction, we just give up and return the instruction.

We were passed the system memory as a pointer to an array of *uint8* - that is, bytes; to keep the code simple, we create a new *uint16* * pointer to read instructions out of memory in a straightforward manner (using *iptr* as an index into the memory). Note that we *always* increment *iptr* when we read the instruction, so the branch distances are one less than one might imagine.

A real interpreter would have a few more bells and whistles - for example, we probably want to change the *model()* function to be able to execute a specified number of instructions, returning when it's done so; this way we can put a simple event-driven user interface around the model - create a forever loop which

- polls the keyboard for a character, and if one has been pressed, act appropriately
- calls the model, specifying that it should execute (say) a million instructions

The 'act accordingly' on the keyboard poll could be to investigate the state of the machine; to single-step the machine; to dump memory; to load memory. . and so forth. If we manage to run the interpreter at 10 MIPS, we'll get to 'interfere' with the simulated machine about as quickly (~100msec) that a human can react. (Putting the poll of the keyboard into the execution loop would slow the system considerably).

We might also need or want to have the machine step one instruction at a time, when debugging the architecture.

From this, we can see that an 'architecture compiler' has a fairly straightforward job. It will take an architecture description, and generate a nested switch statement, extracting the value of the instruction field it's switching on. When it gets to generating code for a specific instruction, it will generate a case element which has a block of code inside matching curly braces, so each instruction can have its own local variables. Each instruction has a variable for each field used in the instruction, like *rd* or *rs*. having read the fields into the variables, these are then used to implement the instruction's *action* - read and add registers, or branch,. . .

2.2. An Example ADL Description

We can imagine something like the following piece of code as a model for a partial ADL description for a our 16 bit architecture:

```
machine_name "archTest";
version "0. 1";
date "January 12 2017";
instruction_length 16;
registers {
  gpr[32][16];  // gprs are 32 bits wide, and
                // there's 16 of them
  iptr[32];   // the iptr is 32 bits wide
  stop[1];    // the stop bit is 1 bit wide
  //iptr[32]; // the iptr is 32 bits wide -- ERROR
  }

fields {
  // the name of a field
  fmt[15:14] opcode;    // unsigned value
  op6[13:8] opcode;
  op2[13:12] opcode;
  SL8[11:4] sval;    // signed value
  BD8[11:4] displacement;
  SL12[11:0] sval:
  BD12[11:0] displacement;
  rd[3:0]  regsel;    // register selector
  rs[7:4] regsel;
  //SL20[19:0] sval:    // ERROR
}

instructions {
  // ------------------- format 0 --------------------

  fmt = 0 {

    op6 = 63: "stop" {            // assembler instruction definition
      instr_count = instr_count - n;   // action when executed
      }
    op6 = 1: "add rd, rs" {
      gpr[rd] = gpr[rd] + gpr[rs];
      }
    op6 = 2:  "sub rd, rs" {
      gpr[rd] = gpr[rd] - gpr[rs];
      }
    op6 = 3:  "mul rd, rs" {
      gpr[rd] = gpr[rd] * gpr[rs];
      }
    op6 = 4:  "div rd, rs" {
      gpr[rd] = gpr[rd] / gpr[rs];
      }
```

```
    op6 = 5:  "rem rd, rs" {
      gpr[rd] = gpr[rd] % gpr[rs];
      }
      // many more instructions
    op6 = 58:  "ld8 rd, rs" {
      uint32 EA = gpr[rs] ;
      gpr[rd] = readMem(memory, 8, EA);    // read 1 byte
      }
    op6 = 59:  "ld16 rd, rs" {
      uint32 EA = gpr[rs] ;
      gpr[rd] = readMem(memory, 16, EA);    // read 2 bytes
      }
    op6 = 60:  "ld32 rd, rs" {
      uint32 EA = gpr[rs] ;
      gpr[rd] = readMem(memory, 32, EA);    // read 4 bytes
      }
      // many more loads and stores
    }

  // ------------------- format 1 --------------------

  fmt = 1 {
    op2 = 0:  "bne rd, BD8" {
      if (gpr[rd] != 0)  iptr += BD8;
      }
    op2 = 1: "ldc rd, SL8" {
      gpr[rd] = SL8;
      }
    }

  // ------------------- format 2 --------------------

  fmt = 2 {
    op2 = 0: "jmp BD12" {
      iptr += BD12;
    }
  }
}

// initial conditions
initial {
  iptr = 0;              // start execution at location 0
  stop = 0;              // don't stop
  }

// the execute-forever loop that the machine performs
operate {
  instruction = readMem[2, iptr];    // read the next instruction
  iptr += 2;
  execute(instruction);        // decode and execute
  }

// when to stop execution
halt {
  stop == 1;
  }
```

Codepiece 2. Example ADL[10]

This description is simply that of what the machine has as resources and does when it executes the instructions. There's nothing in this description which says what the assembler-level mnemonics might look like, nor anything that explicitly tells a compiler generator that an instruction with fmt(0), op7(0) means 'add'.

10. The book's text will contain only snippets of code, very rarely (if ever) all the code involved in a package. But all code is available as a collection of zipped Xcode projects at kivadesigngroupe. com.

A few points of interest: an instruction encoding is composed of a number of *fields*. Some fields will be part of defining what the instruction *does* - for example, add rather than multiply. Other portions are opernds, which can be constant values, or register selectors. The constants can be literal constants, and can be signed or unsigned; or they can be computed values, like branch displacements. The pseudocode declares names for the fields, what their extents (bit starting position, bit ending position) are, and whether they are opcode components, register selectors, literals (signed or unsigned) or displacements.

However, looking at this text, it's clear that any ADL tool trying to understand the definition will need to:

- read the text and extract a 'token'[11]

- if the token is a keyword (like *machine_name* or *fields*) start processing text with code which understands what's expected (we expect a string followed by a semicolon if the keyword is *machine_name*, for example).

- keep track of the various names introduced - like the names *gpr* and *iptr* - and keep track of their characteristics

- read and understand the instruction definitions

- and always complain if something naughty is attempted (for example, the gprs must always be indexed by a value which is a *regsel*).

These are more or less the things any programming language needs to do, so we may as well think of our ADL compiler as a more or less ordinary compiler, one of whose outputs will be an interpreter (written in C) which will behave like our computer (in the sense that, pointed at some memory with program and data loaded into it, it will 'execute' the program and produce the definitive results.)

We will look later at how to add automatic assembler and compiler generation.

2.3. The Tokeniser

Our first task in writing the ADL compiler will be to create code which reads the text of the ADL file and returns the next *token*. A token is a chunk of text which is meaningful to the language we're compiling - it's clear that we have meaningful punctuation, magic words (like *halt*), numbers, comments and more.

The code that does this work is often called a *lexer*. There are tools available to create lexers, but for the sorts of language we're interested in, it's easy enough to write them by hand. We'll call our lexer our *tokeniser.*

Our tokeniser will eventually want to be usable for tokenising multiple files, included nested *include* files, but for the moment we can keep those thoughts in the back of our mind.

But what does it have to do?

The tokeniser has no idea of the meaning of what it's reading; its job is simply to read the next interesting item and deliver it to the code that called it. The code that calls it, in our compiler, is the *parser*. The parser knows what the language is, and has certain expectations about what it will see next - for example, if it's just been told that the token just read was a 'name' whose value was 'halt', it expects the next token to be a left curly brace, '{'. If the next token isn't a left curly brace, then there's a *syntax error* in the ADL, and the parser needs to report this to the user with a helpful error message.

It's most helpful if the tokeniser reports back a sort of label for what it's just found; rather than saying "I found a semicolon", because sometimes it will find a name, or a number. The simplest way of managing this in C is to have an *enumeration* of 'labels' for the various sorts of token we can find. Thus we can find a *t_semicolon, t_name* or *t_intnum*, and when called, the tokeniser will return such a label to the caller. All needed labels will be declared in a header file shared with the calling code.

Since we can see from this description that it's only the tokeniser which sees the raw text in the file, the tokeniser has to maintain sufficient information that the parser can report where in the file what error occurred. Since it reads characters one by one from the file, it needs to build up the current token character by character. It needs to keep track of new lines, and be able to count the lines (so we can have an error message along the lines of "*Error on line 324: expected '{', but saw 'poodle'.* "). Ideally, the error message will be able to print the offending line and indicate exactly which instance of the text 'poodle' the parser is complaining about. So it needs to keep track of what character position in the current line the current token started at. Since it needs to read characters from a file, it also needs to keep the FILE * descriptor to hand, and usefully the filename.

Looks like a *struct* looking somewhat like this should do the job:

```
typedef struct {
    // the file we're processing
    char * fileName;
```

*11. A **token** is a continuous chunk of text, like 'gpr' or 'operate' or '{'; the boundaries between tokens are specified by rules. A name could be any sequence of characters that starts with a letter and continues until a not-letter character is found, for example.*

```
    FILE * src;

    // the most recent token
    char * tokPtr;        // the actual token
    uint tokBuffLength;       // the maximum size of the token buffer
                    // (dynamically increased when necessary)
    uint tokLen;          // where we are in the token buffer
    Token tokType;            // the token type (t_num etc)
    TokErr tokErr;            // any error

    // file housekeeping
    uint32 lineNumber;        // line number in the source
    uint32 charPos;       // current position in the line

    // behavior
    int detectStrings;        // if true, we will detect strings else not
    int detectComments;       // if true, we will detect comments
                    // (// and /*. . */). .  else we won't
    char legal_name_chars[32];

    // token pushback - if true, reuse the token in here
    int unGotToken;

    // if ungotch is non-zero, it's the pushed back character
    char ungotch;
} tokStruct;
```

Codepiece 3. The private data structure for the tokeniser

The tokeniser is a relatively simple piece of code. At the top level, it looks like this:

```
Token tokGetToken(tokData * td) {
    // we pass in a pointer to the user token data structure we want to use
    // the tokeniser writes info back into the other fields
    int ch;
    // find our private data structure
    if (td == NULL) return t_error;
    tokStruct * ts = td -> tokenPtr;

    // did we put the token back?
    if (ts -> unGotToken) {
        ts -> unGotToken = FALSE;
        return ts -> tokType;
    }

    // eat the whitespace
    ch = tokSpaces(ts);

    // beginning of a token
    ts -> tokLen = 0;
    ts -> tokPtr[ts -> tokLen] = ch;
    ts -> tokErr = tokerr_none;
    ts -> tokType = t_none;

    if (ch == EOF) {
        // end of file
        ts -> tokType = t_eof;
    }
    else if (ch == '#')               GetPP(ts, ch);
    else if (isalpha(ch) || (ch == '_'))   GetAlNum(ts, ch);
    else if (isdigit(ch))         GetNumber(ts, ch);
    else if (ispunct(ch))         GetPunct(ts, ch);
    else  {
        ts -> tokType = t_unrecognised;
    }
```

```
        // tidy up the token text buffer
        ts -> tokPtr[ts -> tokLen] = '\0';

        // point td at the token buffer
        td -> tokPtr = ts -> tokPtr;
        return ts -> tokType;
    }
```

Codepiece 4. Top Level of the tokeniser.

The parser has to initialise the tokeniser; when it does this, what's returned is a simpler data structure, *tokData,* containing the token's text, its length, its type, and a pointer to the private *tokStruct* used by the tokeniser and described above. When actually using the parser, we will often have to check that the next token is an opening brace, or a semicolon, or a string, or an integer, etc; so the tokeniser package contains a number of utility routines to simplify this somewhat.

2.4. The simpleADL Compiler

To get our feet wet, we create a simple ADL compiler which will process the example ADL file we've shown above and generate an interpreter (and assembler). The first steps are simple enough - we create a structure with fields corresponding to the elements of the ADL description, thus:

```
    typedef struct {
        char * machine_name;
        char * version;
        char * date;
        uint32 instruction_length;
        char * registers;    // placeholders. . .
        char * fields;
        char * instructions;
        char * initial;
        char * execute;
        char * stop;

    } Architecture;
```

Codepiece 5. Initial version of the structure defining an architecture

Some portions of this structure are easy to fill in, but register, fields, instructions etc will be more complicated; in our first step we'll leave them declared (as above) as character strings as placeholders.

Now we can show the code to eat the ADL definition and build up an instance of this structure:

```
int main(int argc, const char * argv[]) {
    // insert code here. . .
    char * sourceName = (char *) argv[1];
    tokData * td;
    uint64 t = 0;
    Architecture * arch;
    gReport = 0;

    printf("\nsimpleADL [%s] using %s\n", adl_name_and_version, tokNameAndVersion());

    td = tokNewTokeniser(sourceName);
    arch = malloc(sizeof (Architecture));
    if (td && arch) {

        tokDoDetectStrings(td);
        tokDoDetectComments(td);

        t = timenow();
        getMachineName(arch, td);
        getVersion(arch, td);
        getDate(arch, td);
        getInstructionLength(arch, td);
        getResources(arch, td);
```

```
        getFields(arch, td);
        getInstructions(arch, td);
        getInitial(arch, td);
        getOperate(arch, td);
        getHalt(arch, td);

        printArch(arch);

        tokStopTokeniser(td);
        if (gReport) printf("\n\n\t===== end of file ====\n\n");
        t = timenow() - t;
    }
    else {
        printf("\n*** couldn't start tokeniser for file '%s'", sourceName);
    }

    printf("\n\nDone.  Took %lld microseconds\n", t);
    return 0;
}
```

Codepiece 6. The top level of the initial ADL compiler

We simply have a function for each separate chunk of the ADL file, and call them one after another. We can do this because (as currently defined) the ADL file has a specific order; real (ordinary) program is much more demanding.

And the functions are pretty straightforward; as an example, here's what we do to get the machine name:

```
static void getMachineName(Architecture * arch, tokData *td){
    TokErr t = tokerr_none;
    printf("\n getMachineName: ");
    t = tokGetKeyword(td, "machine_name");
    // got the keyword - now get the actual string
    t = tokGetStringToken(td);
    if (t != tokerr_none) {
        adlError(td, "expected a string specifying name of architecture");
        }
    tokSaveTokString(td, &arch -> machine_name);
    tokGetThisToken(td, t_semicolon);
}
```

And when we run the program, we get[12]:

```
simpleADL [simpleADL 1. 0v0] using kiva tokeniser 1. 0v9 January 2017

 getMachineName:
 getVersion:
 getDate:
 getInstructionLength:
 getResources:
 getFields:
 getInstructions:
 getInitial:
 getOperate:
 getHalt:

Architecture Definition:

        Name:             "basic1"
        Version:          "0. 1"
        Date:             "January 12 2017"
        Instruction Length  24
         --- other aspects ---

Done.  Took 799 microseconds
Program ended with exit code: 0
```

12. *Note that there may be minor differences in output and program text between the examples and the released code versions. program output is taken from Xcode's Console/Debug area; code is generally taken from an Xcode edit window*

If we deliberately include an error in the ADL file, we can see what error-reporting looks like; to get the following, just change the instruction length from 24 to "24":

```
simpleADL [simpleADL 1. 0v0] using kiva tokeniser 1. 0v9 January 2017

 getMachineName:
 getVersion:
 getDate:
 getInstructionLength:


**** ERROR:
        in file '/Users/pete/Dropbox/SoftwareProjects–Working/simpleADL/Test/basic. adl' on line
5, character position 24


       Expected 'an integer specifying instruction length' but saw '"24"'
Program ended with exit code: 1
```

Enough to find the error, but not as human-friendly as one would want. We'll fix that at a later time.

Now we should tackle the *resources* and the *fields*. These are more complicated, because *a priori* we don't know how many of each they are, nor what their characteristics are. A simple way of dealing with this is to have a linked list of (for example) register descriptions. As we encounter another register declaration, we create a new register structure, fill in the details, and add it to the list.

A linked list comprises a queue definition structure and per-element structures. We choose to use a doubly-linked list, in which each element in the list has a pointer to its successor and to its predecessor. This makes the addition of elements to the tail of the queue very cheap, and simplifies addition or removal of elements in the 'middle' of the queue. The list of elements is managed by a Queue data structure, which has a pair of pointers, one to the head element and one to the tail. We provide a number of utility routines for managing such queues, which will make frequent appearances in the software we write.

Our list of registers makes use of a common queue structure, a per-element *qBlock* structure to point to next and previous, and the Register structure itself with an embedded qBlock:

```
// the header block for inclusion in each list element
typedef struct block qBlock;

struct block {
    qBlock * next;          // next block
    qBlock * prev;          // previous block
};

// the queue structure, pointing at head and tail of the queue
typedef struct queue {
    char * name;
    qBlock * head;
    qBlock * tail;
} Queue;
// a Register structure, with the queuing block at its base

typedef struct {
    qBlock header;          // queue package header
    char * name;         // name of the register
    uint32 width;           // width in bits
    uint32 count;           // and how many we have
} Register;
```

Codepiece 7. Managing a queue or list of Registers

Now, code to implement the parsing of the *resources* section is pretty straightforward, if rather tedious with the needed error checking:

```
static void  getResources(Architecture * arch, tokData *td){
    TokErr t = tokerr_none;
    Token tok = t_none;
    printf("\n getResources: ");
```

```
        t = tokGetKeyword(td, "resources");
        if (t != tokerr_none) adlError(td, "resources");
        t = tokGetThisToken(td, t_lcb);
        if (t != tokerr_none) adlError(td, "{ in register declaration");

        // now create a Register for each declaration and queue them up on the
        // resources queue
        while (tok != t_rcb) {
            tok = tokGetToken(td);
            if (tok == t_rcb) break;
            if (tok == t_name) {
                // see if we have 'register'
                t = tokCheckToken(td, "register");
                if (t == tokerr_none) {
                    // yep; create a new register
                    Register * r = newRegister();
                    // read a name
                    t = tokGetThisToken(td, t_name);
                    if (t != tokerr_none) adlError(td, "register name");
                    // and save it
                    tokSaveTokString(td, &r -> name);
                    // get the '['
                    tokGetThisToken(td, t_larray);
                    if (t != tokerr_none) adlError(td, "'['");
                    tokGetIntToken(td);
                    if (t != tokerr_none) {
                      adlError(td, "an integer 'size' for the register");
                      }
                    r -> width = (uint32)atol(td -> tokPtr);
                    tokGetThisToken(td, t_rarray);
                    if (t != tokerr_none) adlError(td, "']'");
                    // now see if we get a t_name or a t_larray
                    tok = tokGetToken(td);
                    if (tok == t_semicolon) {
                        // singleton register
                        r -> count = 1;
                    }
                    else {
                        // register file
                        tokGetIntToken(td);
                        r -> count = (uint32)atol(td -> tokPtr);
                        tokGetThisToken(td, t_rarray);
                        if (t != tokerr_none) adlError(td, "']'");
                        tokGetThisToken(td, t_semicolon);
                        if (t != tokerr_none) adlError(td, "';'");
                    }
                    // ok - now queue this up on the registers list
                    qAddBlock(&arch -> registers, (qBlock *) r);
                }
            else {
                adlError(td, "register in register declaration");
            }
            }
        }
    }
```

Codepiece 8. Parsing and remembering the resources declaration in ADL

Similarly, we build a list of Field descriptors.

2.4.1. Checking Names for Uniqueness

With correct ADL, this all works just fine, and the declarations are captured. But we're missing an important correctness check - it's presumably important that we don't use the same name for two different registers, or for two different fields. Checking this is simple enough - once we've collected all the data, just run back down the lists

and see if there's replication. But when there's lots of names, or the names have scope - perhaps it's legal to have fields share names with registers, for example - then it gets messy fast.

A simple way of handling this is to introduce another data structure, the *symbol table*. This is a hash table with queues for each hash value, each hash value queue holding a list of elements which define a name and some properties.

More straightforwardly, the symbol table is a structure containing some useful info, plus an array of the queues we've just seen. The number of queues will vary with the number of unique names we expect to handle,

We add a new entry to one of these queues whenever we see a new name. We choose the specific queue by performing a computation on the name - basically, we eat the all characters in the name and do some shifts and xors to compute a queue index, and then add a symbol structure noting the name to the queue. To see if we already have stored a name, we perform the computation on the name, and run down the appropriate queue looking for a match with an already-stored name; if we have *n* queues, this reduces the average search time (compared to looking through a list of all the names) by a factor of *n*.

To speed things up, we store the length of the string representing the name of a symbol along with the string; then we can reject a match more quickly by checking lengths before doing a string compare.

The computation of which queue to use involves a two-step computation - first, do the hashing operation of xoring and shifting the string to form an unsigned value, say *h* - call this the *prehash* value - then compute the queue by calculating *h* % *queue array size*. So that we can efficiently install a symbol in a symbol table of any size, we compute the prehash value for each name and save it with the symbol.

Here's the symbol table declaration:

```
typedef struct {
    qBlock header;          // so we can queue them up
    char * name;            // table name
    uint32 lexlevel;        // block-structured symbol table
    uint32 size;            // number of entries
    Queue * queues;         // pointer to the n queues
} symTab;
```

And a Symbol:

```
typedef struct {
    qBlock header;          // next struct in the hashchain
    char * name;            // pointer to name
    uint32 len;             // length of the name
    uint32 prehash;         // the prehash
    void * ptr;
} Symbol;
```

Codepiece 9. The symbol table and symbol structures

The Register and Field declaration code now looks a little different; the Register code is as follows (with the new symbol management code bolded):

```
static void  getResources(Architecture * arch, tokData *td){
    TokErr t = tokerr_none;
    Token tok = t_none;
    symTab * st = arch -> symtab;
    t = tokGetKeyword(td, "resources");
    if (t != tokerr_none) adlError(td, "resources");
    t = tokGetThisToken(td, t_lcb);
    if (t != tokerr_none) adlError(td, "{ in register declaration");

    // now create a Register for each declaration
    // and queue them up on the resources queue
    while (tok != t_rcb) {
        tok = tokGetToken(td);
        if (tok == t_rcb) break;
        if (tok == t_name) {
            // see if we have 'register'
            t = tokCheckToken(td, "register");
            if (t == tokerr_none) {
                // yep; create a new register
                Register * r = newRegister();
```

```
                // read a name
                t = tokGetThisToken(td, t_name);
                if (t != tokerr_none) adlError(td, "register name");
                tokInfo(td);
                // look it up in our symbol table
                Symbol * sym = symLookup(st, td -> tokPtr);
                if (sym) {
                    // curses - we've seen this name before!
                    adlError(td, "unique register name");
                }

                // add the name to the symtab
                sym = symInstallName(st, td -> tokPtr);

                // link the symbol to the Register
                sym -> ptr = r;
                // and the register to the sym
                r -> sym = sym;
                r -> name = strdup(td -> tokPtr);

                // get the '['
                …       // omitted for space reasons
                // ok - now queue this up on the registers queue
                qAddBlock(&arch -> registers, (qBlock *) r);
            }
        else {
            adlError(td, "register in register declaration");
        }
        }
    }
}
```

Codepiece 10. Using the symbol table to ensure uniqueness of names

2.4.2. Capturing the Instruction Definitions

Looking at the declarations of the instructions, we see they're hierarchically-organised.

```
instructions {
// ------------------- format 0 --------------------
fmt = 0 {
    op7 = 0: "add" {
        gpr[rd] = gpr[rs1] + gpr[rs2];
        }
```

We see the keyword *instructions*, followed by a left curly brace, followed by a field name (here, *fmt*) in the form *fmt = 0*. Then there's another left curly brace, followed by another field name and (this time) a colon, followed by text defining the instruction name and its behaviour and with matching right curly braces. In this example, we're only descending one level before encountering the actual instructions; in a more complex architecture, this nesting could go somewhat deeper.

So we need to create a hierarchical data structure whose end elements will define the actual instructions, and whose intermediate levels will keep track of the common-at-that-level format values. At each level, we'll generally have a number of elements, which we can represent using the queues we've already encountered. Our overall description will be a queue of outer-level structures, each of which points to the head of a queue of one-level down structures, each of which… etc, until we end up at a level of structures which point at individual instruction specifications.

The structures form the backbone of the description, so:

```
typedef struct spine Spine;
struct spine {
    qBlock header;   // so we can queue 'em up
    Symbol * field;  // the name of the field for which this defines a value
    uint64 value;        // the actual value (in the field)
    uint32 leaf;     // 0 if this is a leaf spine, in which case
                // the instruc pointer is valid
```

```
    union {
        Queue * spine;   // there's another level
        InstrDef * instruc;  // last level (the field had a colon)
    } ptr;
};
```

Codepiece 11. A Spine structure to manage instruction definitions

The declaration should be pretty self-descriptive. We'll read the ADL for the instructions in an obvious manner; first, kick the operation off:

```
static void  getInstructions(Architecture * arch, tokData *td){
    TokErr t = tokerr_none;
    t = tokGetKeyword(td, "instructions");
    if (t != tokerr_none) {
        adlError(td, "key word instructions");
    }
    t = tokGetThisToken(td, t_lcb);
    if (t != tokerr_none) {
        adlError(td, "{ at the startinstruction definitions");
    }
    getInstructionBlock(arch, &arch -> spine, td, 0);
}
```

Codepiece 12. Kicking off reading instruction definitions

And then get a level of instruction definition using *getInstructionBlock()*.

```
static void getInstructionBlock(Architecture * arch, Queue * q, tokData * td, uint32 level) {
    /* read a level of instruction definition, from a { to a } and create the appropriate Spine list
    when we hit a lowest level (finding a ':' in place of a '{') go eat the instruction definition instead
    we have already read the {
    */
    uint32 count = 0;
    TokErr t_err = tokerr_none;
    Token t;
    indent(level);
    while (TRUE) {
        // we next expect a field name, or an '}'.  We're done if it's }
        t = tokGetToken(td);
        if (t == t_rcb) {
            return;
        }
        else if (t != t_name) {
            adlError(td, "expected a field name");
        }
        // look up the name
        Symbol * sym = symLookup(arch -> symtab, td -> tokPtr);
        if (sym) {
            Element * element = sym -> ptr;
            if (element -> etype != adlField) {
                adlError(td, "a field name");
            }
            // ok, it's a field; get an =
            t_err = tokGetThisToken(td, t_assign);
            if (t_err != tokerr_none) {
                adlError(td, "'=' in a field value specification");
                }
            t_err = tokGetThisToken(td, t_intnum);
            tokInfo(td);
            uint64 value = atol(td -> tokPtr);
            if (t_err != tokerr_none) {
                adlError(td, "an integer value for a field");
            }
            // create the spine element
            Spine * s = newSpine(sym, value);
            // and add it to the current spine queue
            qAddBlock(q, (qBlock *)s);
```

```
            // look at the next token:
            t = tokGetToken(td);
            if (t == t_colon) {  // it's an instruction definition
                InstrDef * instr = getAnInstruction(arch, td, level+1);
                s -> leaf = 1;
                s -> ptr. instruc = instr;
                // point the spine at the instruction
                // report
                printInstruc(instr, level);
            }
            else if (t == t_lcb) {    // it's another level of definition
                char name[32];
                count++;
                Queue * nextq = qNewQ(name);
                s -> leaf = 0;
                s -> ptr. spine = nextq;
                getInstructionBlock(arch, nextq, td, level + 1);
            }
            else {
                // error
                adlError(td, "a { or a : in instruction definition");
            }
        }
    }
}
```

Codepiece 13. Reading an instruction definition level

When we run this with our example ADL, we see:

```
simpleADL [simpleADL 1. 0v3] using simpleTokeniser 1. 0v9 January 2017

getMachineName:
getVersion:
getDate:
getInstructionLength:
getRegisters:
getFields:


            get an instruction
        instruction "stop" has action

            // assembler instruction definition
            instr_count = instr_count - n;        // action when executed

            get an instruction
        instruction "add rd, rs" has action

            gpr[rd] = gpr[rd] + gpr[rs];

            get an instruction
        instruction "sub rd, rs" has action

            gpr[rd] = gpr[rd] - gpr[rs];

            get an instruction
        instruction "mul rd, rs" has action

            gpr[rd] = gpr[rd] * gpr[rs];

            get an instruction
        instruction "div rd, rs" has action

            gpr[rd] = gpr[rd] / gpr[rs];

            get an instruction
        instruction "rem rd, rs" has action
```

```
        gpr[rd] = gpr[rd] % gpr[rs];

        get an instruction
    instruction "ld8 rd, rs" has action

        uint32 EA = gpr[rs] ;
        gpr[rd] = readMem(memory, 8, EA);     // read a byte

        get an instruction
    instruction "ld16 rd, rs" has action

        uint32 EA = gpr[rs] ;
        gpr[rd] = readMem(memory, 16, EA);    // read 2 bytes

        get an instruction
    instruction "ld32 rd, rs" has action

        uint32 EA = gpr[rs] ;
        gpr[rd] = readMem(memory, 32, EA);    // read 4 bytes


        get an instruction
    instruction "bne rd, BD8" has action

        if (gpr[rd] != 0)  iptr += BD8;

        get an instruction
    instruction "ldc rd, SL8" has action

        gpr[rd] = SL8;


        get an instruction
    instruction "jmp BD12" has action

        iptr += BD12;
```

2.4.3. Generating the Interpreter

We don't have all the information we need to create a complete executable model, but ignoring the specifications for initialisation, fetching and stopping we have enough to generate *something*.

First, let's just print out the architecture as specified so far. The new bit is printing out the instruction definitions, which we'll do by walking the data structure we've built in the previous section. To print the architecture, we use this set of functions:

```
// ------------ printSpine --------------

static void printSpine(uint32 doPrint, Spine * spine, uint32 level) {
    if (spine -> leaf) {
        indent(doPrint, level);
        printf("%s = %lld", spine -> field -> name, spine -> value);
        indent(doPrint, level);
        printInstruc(doPrint, spine -> ptr. instruc, level + 1);
    }
    else {
        indent(doPrint, level);
        printf("%s = %lld", spine -> field -> name, spine -> value);
        Queue * q = spine -> ptr. spine;
        printSpineQ(doPrint, q, level + 1);
    }
}

// ------------- printSpineQ -----------

static void printSpineQ(uint32 doPrint, Queue * q, uint32 level) {
```

```
        // print all the instructions on this queue

        if (doPrint) {
            indent(doPrint, level);
            Spine * spine = qFirst(q);
            while (spine) {
                printSpine(doPrint, spine, level);
                spine = qNext(spine);
            }
        }
    }
// -------------- printArch --------------

static void printArch(Architecture * arch) {
    printf("\n\nArchitecture Definition:\n");
    printf("\n\tName:             %s", arch -> machine_name);
    printf("\n\tVersion:          %s", arch -> version);
    printf("\n\tDate:             %s", arch -> date);
    printf("\n\tInstruction Length  %d", arch -> instruction_length);
    // now do the linked list of registers
    printf("\n\tElements: registers and fields");
    Element * element = qFirst(&arch -> elements);
    while (element) {
        printElement(element);
        element = qNext(element);
    }
    printf("\n\tInstructions:");
    printSpineQ(gReport, &arch->spine, 0);
}
```

indent() is a function which prints the number of tabs specified by its *level* argument. The code does a classic recursive walk of the data structure, eventually hitting *printInstruc()*, which is

```
// -------------- printInstruc -------------

static void printInstruc(uint32 doPrint, InstrDef * instruc, uint32 n) {
    indent(doPrint, n);
    if (doPrint) {
        printf("instruction %s has action\n",
            instruc -> name);
        indent(doPrint, n);
        printf("\t%s",instruc -> act -> string);
    }
}
```

Running this. we get:

```
Architecture Definition:

        Name:             archTest
        Version:          0. 1
        Date:             January 12 2017
        Instruction Length  16
        Elements: registers and fields
            register: gpr[32][16];
            register: iptr[32];
            register: stop[1];
            field: fmt[15:14]: unsigned value;
            field: op6[13:8]: unsigned value;
            field: op2[13:12]: unsigned value;
            field: SL8[11:4]: signed value;
            field: BD8[11:4]: branch displacement;
            field: SL12[11:0]: signed value;
            field: BD12[11:0]: branch displacement;
            field: rd[4:0]: register selector;
            field: rs[7:4]: register selector;
        Instructions:
```

```
fmt = 0

    op6 = 75

        instruction "stop" has action

            // assembler instruction definition
        instr_count = instr_count - n;    // action when executed

    op6 = 1

        instruction "add rd, rs" has action

            gpr[rd] = gpr[rd] + gpr[rs];

    op6 = 2

        instruction "sub rd, rs" has action

            gpr[rd] = gpr[rd] - gpr[rs];

    op6 = 3

        instruction "mul rd, rs" has action

            gpr[rd] = gpr[rd] * gpr[rs];

    op6 = 4

        instruction "div rd, rs" has action

            gpr[rd] = gpr[rd] / gpr[rs];

    op6 = 5

        instruction "rem rd, rs" has action

            gpr[rd] = gpr[rd] % gpr[rs];

    op6 = 64

        instruction "ld8 rd, rs" has action

            uint32 EA = gpr[rs] ;
        gpr[rd] = readMem(memory, 8, EA);     // read a byte

    op6 = 65

        instruction "ld16 rd, rs" has action

            uint32 EA = gpr[rs] ;
        gpr[rd] = readMem(memory, 16, EA);    // read 2 byte2

    op6 = 66

        instruction "ld32 rd, rs" has action

            uint32 EA = gpr[rs] ;
        gpr[rd] = readMem(memory, 32, EA);    // read 4 bytes

fmt = 1

    op2 = 0

        instruction "bne rd, BD8" has action

            if (gpr[rd] != 0)  iptr += BD8;
```

```
        op2 = 1

            instruction "ldc rd, SL8" has action

                gpr[rd] = SL8;

    fmt = 2

        op2 = 0

            instruction "jmp BD12" has action

                iptr += BD12;
```

Looks good. To create an executable model, we walk the structures in the same way and output to a pair of files, *model.c* and *model.h*.

model.c contains the function *model()* which is the actual interpreter; it will look a lot like our handwritten approximation shown earlier. We create a bunch of *#define*s for the fields - *fmt*, for example, is:

```
#define fmt ((instruction >> 14) & 0x00000003)     // unsigned value;
```

We need to sign-extend the fields which are signed values or branch displacements. We write a function *sext(int n, uint64 v)* which looks at an unsigned value's bit *n*, and if that bit is set whacks a bunch of one's into the top of the value. A signed field will have a definition like this:

```
#define SL8 sext(8,((instruction >> 4) & 0x000000ff))      // signed value;
```

model.h contains necessary declarations (such as for model()).

2.5. What's Missing?

2.5.1. A Run-time for the simulator

The ADL tool generates a pair of files which form the executable model of the architecture. But we also need some infrastructure - shared between architectures - which sets up the environment for the simulation, loads code and data, and exercises the model appropriately.

This infrastructure code might be of two forms

- one, interactive, allowing a user to exercise the model interactively, looking inside at state and so forth;
- and the other of a batch form. This might use command-line arguments to configure itself, or eat a file of commands (equivalent to the UI commands of the interactive runtime).

We could construct an interactive runtime using the *ncurses* library. This avoids the use of a full-blown graphical UI and is portable to any UNIX-y system. It allows the placing of text anywhere in a window, and the polling of the keyboard (for example, to stop a runaway program). Because it has to poll for keyboard input, its overall efficiency will be lower than the batch runtime.

We don't show the interactive code here.

2.5.2. A Program Loader.

Our runtime needs to be able to load code and data into our simulated memory.

Before looking at that, let's assess where we are - we're able to create a chunk of code which looks a lot like an interpreter, but which works properly only when there are no errors whatsoever in the adl. In particular, we never

- capture what 'variable' fields the instruction has (e.g., we don't capture that an add must have an rd field and an rs field)
- check that, for example, an opcode fits in its specified field
- check the action is legal code and uses only legal resources

Fixing this is just more code, and we will deliberately and wickedly put the task to one side for the moment while we begin to look at the problem of writing software (in assembler) for our machine. A first step would be able to load stuff into the machine's memory; at this stage we don't need anything sophisticated, so we'll first create a loader format - a format for loadable files.

A loadable file has to specify

- code to be put into memory - eventually, this will be executable
- data to be put into memory - eventually, this will be read only

- where to start execution

And we want it human-readable, so the values should be hex values.

So we can end up with something rather straightforward, like this:

```
title example. ldr
codeseg 100 100
dataseg 200 50
start 100

// a real program
memory 150
code
    0x50 0x01
    0x50 0x82
    0xc1 0x11
    0x06 0x23
    0x02 0x13
    0x4f 0xa3
    0x3f 0x00
end

// just checking the keyword
codeat 150
1a 1b 1c 1d 1e 1f end
//data isn't used
data
1 2 3 4 5 6 7 8  end
string b hello world
stop
```

Codepiece 14. Example Simple Loader Format

The loader format uses a small vocabulary of keywords to specify the memory layout, contents, and the address at which to start execution.

- *title* gives a name to the program
- *codeseg* specifies the base and size of the code segment
- *dataseg* does the same for the data segment
- *memory* specifies the necessary memory size (in KB)
- *start* specifies the address of the instruction at which execution should start
- *code* says 'write these bytes into the code segment, starting where we last stopped, or at the base of the code segment otherwise'
- *codeat* says the same, but specifies the address at which the bytes must start to be written.
- *data* says 'write these bytes into the data segment, starting where we last stopped, or at the base of the data segment otherwise'
- *string* is followed by a size and then an unquoted string. It's to be placed in the next available spot in the data segment
- *end* says that we've reached the end of the loader file

All the values are hex values; they can start with 0x, or not, as desired. At this stage, we don't worry about relocations; we assume the creator of the loader file has created a correct memory image, and just stuff it into simulated memory.

Oh, what's a *segment*? It's a contiguous chunk of address space (memory addresses) which has a single set of properties. A code segment, in a sensible machine, contains just executable instructions, and can only be executed, nor read or written by program. A data segment contains data, and (in the same sensible machine) can perhaps be read and written, or just read, or just written, but none of its contents can be executed. Our architecture doesn't know (at the moment) anything about these access permissions, so the code segment is just the range fo addresses where we pile up code (if the assembler source intersperses data and program), and the data segment is where we pile up data.

Our loader implementation is of the el-cheapo variety. The vocabulary is so limited there's no need to use a symbol table and our proper tokeniser. Instead, we use a very simple tokeniser, which reads characters from a specified file into a global, character string gToken[], and does a sequence of comparisons to return an appropriate enum to the caller.

```
// ------------------- get_token --------------------

Token get_token(FILE * f) {

    uint n = 0;
    char c = ' ';
    // eat whitespace
    while ((c != EOF) && (isspace(c))) {
        c = fgetc(f);
        if (c == '/') {
            c = fgetc(f);
            if (c == '/') {
                // comment - read to end of line
                while ((c != '\r') && (c != '\n')) c = fgetc(f);
            }
            ungetc(c, f);
            c = '/';
        }
    }
    while (isalnum(c)) {
        gToken[n++] = c;
        c = fgetc(f);
    }
    gToken[n] = '\0';
    gTokLen = n;
    if (strcmp("code", gToken) == 0) return t_code;
    else if (strcmp("title", gToken) == 0) return t_title;
    else if (strcmp("codeat", gToken) == 0) return t_code_at;
    else if (strcmp("data", gToken) == 0) return t_data;
    else if (strcmp("start", gToken) == 0) return t_start;
    else if (strcmp("codeseg", gToken) == 0) return t_codeseg;
    else if (strcmp("dataseg", gToken) == 0) return t_dataseg;
    else if (strcmp("memory", gToken) == 0) return t_memory;
    else if (strcmp("string", gToken) == 0) return t_string;
    else if (strcmp("end", gToken) == 0) return t_end;
    else if (strcmp("stop", gToken) == 0) return t_stop;
    else return t_value;
}
```

Codepiece 15. The loader tokeniser.

The loader itself is pretty simple and obvious. Its complications, such as they are, arise from some simple desires:

- We don't want to load code if the code segment hasn't been defined, or if the code is outside the segment.
- We don't want to execute code if we don't have a start address or if we haven't loaded code.
- and so on

It looks like this:

```
uint8 * loader(FILE * f, LoadRecord * lr) {

    /* expect a file of this general shape:

        code
        hexval hexval hexval hexval hexval hexval hexval hexval hexval....end
        data
        hexval hexval hexval hexval hexval hexval hexval hexval hexval....end
        start    hexval
        codeseg  hexval hexval
        dataseg hexval hexval
        memory   hexval
        string   hexval <character string>  (note, no quotes. No terminating '\0'. That's added in loading
     */
    Token t = t_code;
    uint64 ptr = 0;
    uint64 code_base = 0, data_base = 0, code_size = 0, data_size = 0, mem_size = 0;
    uint64 code_top = 0, data_top = 0, exec_start = 0;
    SegType s = seg_none;
```

```
    if (gLoadReport) printf("\n\t--- Loading");
    uint32 status = 0;
    while (t != t_stop) {
        t = get_token(f);
        switch (t) {
            case t_title:
            {   char c = fgetc(f);
                if (gLoadReport) printf("\nprogram name: '");;
                while ((c != '\r') && (c != '\n')) {
                    if (gLoadReport) printf("%c", c);
                    c = fgetc(f);
                }
                if (gLoadReport) printf("'");
            }
                break;

            case t_code:
                // set the pointer to the top of the code segment
                if (gLoadReport) printf("\ncode: load into code segment from code_top(0x%08llx)",
code_top);
                if (status & g_codeseg) {
                    if (s == seg_data) {
                        data_top = ptr;
                    }
                    s = seg_code;
                    ptr = code_top;
                    // read values until the token isn't a 'value' but is an 'end'
                    ptr = load_bytes(f, lr -> memory, ptr);
                    status |= g_code;
                }
                else {
                    printf("Loader error - code segment not defined.");
                }
                break;

            case t_code_at:
                // we have a hex value following the token. set ptr to that
            {
                uint64 ptr2;
                if (gLoadReport) printf("\ncode at:");
                t = get_token(f);
                if (t != t_value) load_err("expected code load address");
                else {
                    ptr2 = strtol(gToken, 0, 16);
                    if (gLoadReport) printf("\tload to address 0x%08llx", ptr2);
                    if ((ptr2 >= code_base) && (ptr2 < code_base + code_size)) {
                        // ok - load the code
                        load_bytes(f, lr -> memory, ptr2);
                        status |= g_code;
                    }
                    else {
                        printf("Loader error - load address (%lld) outside code segment (%lld..%lld,
0x%08llx..0x%08llx).",
                            ptr2, code_base, code_base + code_size, code_base, code_base + code_size);
                    }
                }
            }
                break;

            case t_data:
                // set the pointer to the top of the data segment
                if (gLoadReport) printf("\ndata:");
                if (status & g_dataseg) {

                    if (s == seg_code) {
```

```
                code_top = ptr;
            }
            s = seg_data;
            ptr = data_top;
            status |= g_data;
                // read values until the token isn't a 'value' but is an 'end'
            ptr = load_bytes(f, lr -> memory, ptr);
        }
        else {
            printf("Loader error - data segment not defined.");
        }
        break;

    case t_start:
        // store the start address
        if (gLoadReport) printf("\nstart address=");
        t = get_token(f);
        if (t != t_value) load_err("expected execution start address");
        exec_start = strtol(gToken, 0, 16);
        if (gLoadReport) printf("\t0x%08llx", exec_start);
        status |= g_start;
        break;

    case t_codeseg:
        // get codeseg start and address
        if (gLoadReport) printf("\ncodeseg:");
        t = get_token(f);
        if (t != t_value) load_err("expected segment start address");
        code_base = strtol(gToken, 0, 16);
        t = get_token(f);
        if (t != t_value) load_err("expected segment size");
        code_size = strtol(gToken, 0, 16);
        code_top = code_base;
        if (gLoadReport) printf("\tstart=0x%08llx size=0x%08llx", code_base, code_size);
        status |= g_codeseg;
        break;

    case t_dataseg:
        // get dataseg start and address
        if (gLoadReport) printf("\ndata segment:");
        t = get_token(f);
        if (t != t_value) load_err("expected segment start address");
        data_base = strtol(gToken, 0, 16);
        t = get_token(f);
        if (t != t_value) load_err("expected segment size");
        data_size = strtol(gToken, 0, 16);
        data_top = data_base;
        if (gLoadReport) printf("\tstart=0x%08llx size=0x%08llx", data_base, data_size);
        status |= g_dataseg;
        break;

    case t_memory:
        // get memory size
        if (gLoadReport) printf("\nmemory");
        t = get_token(f);
        if (t != t_value) load_err("expected memory size required");
        mem_size = strtol(gToken, 0, 16);
        status |= g_mem;
        lr -> mem_size = mem_size;
        lr -> memory = malloc(mem_size);
        // randomize the content of memory
        int i;
        for (i = 0; i < lr -> mem_size; i++) {
            lr -> memory[i] = ((i+5)*(i-7)) & 255;
        }
```

```
                break;

            case t_string:
                {
                // get a string. first, get length so we know when to stop
                uint64 count = 0;;
                t = get_token(f);
                if (t != t_value) load_err("expected string length");
                count = strtol(gToken, 0, 16);
                // then get that number of characters
                while (count != 0) {
                    char c = fgetc(f);
                    lr -> memory[ptr++] = c & 255;
                    count--;
                }
                lr -> memory[ptr] = '\0';
                }
                break;

            case t_value:
                // insert a sequence of byte hexvalues into memory at mem[ptr].
                while (t != t_end) {
                    lr -> memory[ptr++] = strtol(gToken, 0, 16);
                    t = get_token(f);
                }
                if (s == seg_code) status |= g_code;
                else if (s == seg_data) status |= g_data;
                break;

            case t_stop:
                lr -> code_base = code_base;
                lr -> code_size = code_size;
                lr -> data_base = data_base;
                lr -> data_size = data_size;
                lr -> start = exec_start;
                lr -> status = status;
                break;

            default:
                load_err("unexpected token");
        }
    }
    return lr -> memory;
}
```

Codepiece 16. The Loader

2.5.3. Looking Inside the Processor State

From time to time, such as when single-stepping, we will want to be able to see what state the processor is in. To do this, we need to print out its context. But the printing will be different for different simulation environments - in the interactive runtime, we use mvprintw() to place text where we want in the window, while in the batch runtime we simply use printf().

Since the context is different for every architecture, so it seems simpler to have the adl compiler generate accessor functions which can be used to fill up an array of register descriptors with the contents of the context; then the environment can display the register descriptors as they wish.

The method we use is the *get_state()* function, generated by ADL, and included in the *model. c* file; it looks like this for our baby architecture:

```
typedef struct {
    char name[32];
    uint64 val;
    } regVal;
```

```
uint32 get_state(Context * context, uint64 n, regVal * r) {
    switch(n) {
        case 0: sprintf(r -> name, "r0");
            r -> val = context -> r[0];
            return 1;
        …   // all the other GPRs
        case 15: sprintf(r -> name, "r15");
            r -> val = context -> r[15];
            return 1;
        case 16:
            sprintf(r -> name, "iptr");       // singleton
            r -> val = context -> iptr; return 1;
        case 17:
            sprintf(r -> name, "stop");       // singleton
            r -> val = context -> stop; return 1;
        case 18:
            sprintf(r -> name, "instruction");
            r -> val = context -> instruction;
            return 1;
        default: return 0;
        }
    }
```

Codepiece 17. The *get_state()* function

Its usage is straightforward; we just repeatedly call *get_state()* specifying the context and an incrementing register index; *get_state()* fills in the *regVal* appropriately, and we print it. We stop when *get_state()* says it has no more registers to describe, by returning *0* instead of *1*:

```
        // show regs
        regVal reg;
        uint32 n = 1, r = 0;
        n = get_state(c, r, &reg);
        while (n) {
            printf("\n\t\t%11llu\t0x%08llx -> %s",
                reg. val, reg. val, reg. name);
            r++;
            n = get_state(c, r, &reg);
            }
```

Codepiece 18. Using *get_state()*

2.5.4. An Assembler

Eventually, we will have a compiler, which will (eventually) emit architecture-specific assembler code. Then we need a per-architecture assembler to convert that to loadable bytes - and the ADL compiler will generate the assembler.

Our first-attempt assembler will be simple enough. There's a common infrastructure (common across architectures) plus a per-architecture portion. The per-architecture portion will recognise the assembler mnemonics and instruction formats define in the architecture ADL; the common infrastructure will manage the representation of the program and its eventual transformation into loadable format.

The first thing we do is to create by hand an assembler for the architecture we're playing with.

In doing this, we'll be conscious that we want it to be driven by architecture-specific data - ideally, data structures which would be generated by the ADL compiler. The idea is to write a generic assembler which is customised for a specific data structure by the declaration of a small number of architecture-specific data structures - like the fields we've defined, and the encodings of the instructions. We'll pretty quickly see that our laziness in the ADL compiler of using a string to specify the mnemonic and operands:

```
        op6 = 1: "add rd, rs" {   // assembler instruction definition
            r[rd] = r[rd] + r[rs];    // action when executed
            }
```

just doesn't do it - we need to actually parse and understand the mnemonic to create the information needed by the assembler - in fact, the natural representation for the assembler is a mnemonic followed by a list of fixed fields and their values, along with a list of operand values (here, we see that the mnemonic 'add' takes two operand fields, rd and rs).

So we'll go back to the ADL compiler and add-in the collection and storage of this info.

The code we used to have looked like this - it fetched a string and saved it into the instruc's name field

```
            t_err = tokGetThisToken(td, t_string);
            if (t_err != tokerr_none) {
                adlError(td, "a string giving the mnemonic");
                }
            // save it
            instruc -> name = strdup(td -> tokPtr);
```

We extend the definition of an instruction definition to hold a pointer to a symbol representing the mnemonic, and to hold an operand count and a queue of operands:

```
typedef struct {
    qBlock header;
    Symbol * mnem;   // the symbol for the instruction name, like "addi"
    Symbol * field;  // the fixed field it defines, like op2
    uint64 value;        // the value of that field
    Action * act;        // pointer to what will be the tree defining the action
    int32 numOpnds;  // the number of operands that the instruction has
    Queue opnds;     // a queue of operands.
                // The queue comprises Opnd structures
} InstrDef;
```

Each operand is represented by an Opnd structure:

```
typedef struct {
    qBlock header;
    Symbol * sym;
} Opnd;
```

And the new code in *getAnInstruction()* is:

```
        // get the instruction mnemonic
        if (instruc) {
            // we expect a new name as mnemonic,
            // followed by a comma-separated list of field names
            tok = tokGetToken(td);
            if (tok == t_name) {
                // looking good - see if it exists
                Symbol * mnem = symLookup(arch -> symtab, td -> tokPtr);
                if (mnem) {
                    // error - the mnemonic exists
                    adlError(td, "unique mnemonic name");
                    return NULL;
                }
                // install the menemonic
                mnem = symInstallName(arch -> symtab, td -> tokPtr);
                printf("\n\tfound a mnemonic '%s'", mnem -> name);
                // ok, unique mnemonic.  Add it in to this instruction
                instruc -> mnem = mnem;
                mnem -> ptr = instruc;
                // now we need to get operand fields.
                // Pattern is field, field,. . .  terminated by the {
                while (TRUE) {
                    // get what we hope is a field name
                    tok = tokGetToken(td);
                    if (tok == t_lcb) {
                        // we take no operands!
                        break;
                    }

                    else if (tok != t_name) {
                        adlError(td, "field name");
                        return NULL;
                    }
                    printf("\n\tfound name '%s'", td -> tokPtr);
                    // goody.  Look it up
                    Symbol * field = symLookup(arch -> symtab,
```

```
                    td -> tokPtr);
            if (field == NULL) {
                adlError(td, "field name");
                return NULL;
            }
            printf("\t - goody, it exists in the symbol table");
            Opnd * op = newOpnd(field);
            qAddBlock(&instruc -> opnds, (qBlock *) op);
            instruc -> numOpnds++;

            // the next token should be a comma (more operands)
            // or an lcb (no more)
            tok = tokGetToken(td);
            if (tok == t_lcb) {
                // we're done with operands
                break;
            }
            else if (tok != t_comma) {
                adlError(td, "',' to separate operands");
                return NULL;
            }
        }
    }
    // we have read all the operands and consumed the '{'
    printf("\ninstruc %s has %d operands",
            instruc -> mnem -> name, instruc -> numOpnds);
    ...
```

Codepiece 19. Reading an instruction's mnemonic and operand list

3. The Assembler

An assembler is, like our other tools, a relatively simple program. There's a small number of keywords plus the details of the instruction set and its *mnemonics* (the mnemonics are the textual representation of the operation, like 'addc').

To design our assembler-generator, we'll first write an assembler from scratch, keeping an eye on the information that must be generated by the ADL compiler, and then return to the ADL compiler and add the assembler-generator code.

First, let's write the assembler.

3.1. A Stand-Alone Assembler

We need to know everything about an instruction definition. This means we need to know its encoding - what fields does it include, and what are their values? What's the mnemonic? And so forth.. Looks like a structure like this can capture everything we need:

```
typedef struct {
    Symbol sym;
    instrucVal val;
    uint32 numOpndFields;     // the number of operand fields
    fieldSel OpndFields[maxFields];   // the operand fields (rd, SL8,..)
    uint32 numFixedFields;        // the number of fixed fields
    fieldSel fixedFields[maxFields];  // the fixed fields (fmt, op6..)
    uint32 fieldValues[maxFields];    // the values for those fields
    } InstrucDef;
```

Codepiece 20. The assembler's instruction definition structure

We have a Symbol for each mnemonic, and will need to declare all the mnemonics to get them into the symbol table.

Now, just to confuse matters, **we're going to change the definition of a Symbol** from the one we used in the ADL compiler. That Symbol was of this form:

```
typedef struct {
    qBlock header;    // next struct in the hashchain
    char * name;      // pointer to name
    uint32 len;       // length of the name
```

```
    uint32 prehash;  // the prehash - we've done the computation but not
                     // reduced to span the symtab size
    void * ptr;      // we will need to link the Symbol with some other
                     // entity - ptr indicates which one
} Symbol;
```

If we used this form, we'd have a pointer inside our InstrucDef pointing at its Symbol, and the *ptr* field in the Symbol would point at our InstrucDef. It's perfectly generous, but - we need indirections to get from one to the other. And somewhere, we need in the thing the *ptr* points at, an indication of exactly what that thing is (when we're reading source from a file, and match the symbol, we use the pointer to find the linked entity. But what is it?). So we need a standard 'protocol' defining all the sorts of things those entities can be.

This is much more simply done by redefining Symbol as:

```
typedef struct {
    qBlock header;   // next struct in the hashchain
    char * name;     // pointer to name
    uint32 len;      // length of the name
    uint32 prehash;  // the prehash - we've done the hashing computation but
                     // not reduced to span the symtab size
    symTag tag;      // a per-symbol value intended to indicate type,
                     // understood only by the using app
} Symbol;
```

Codepiece 21. Redefined Symbol data structure

We make two important changes:

- The ptr field is replaced by value we call *tag*. The application using the symbols package has to provide an enumeration specifying the values that the symTag can take; for example, in the assembler we will have tag values to differentiate between Symbols used for fields, keywords, instructions, labels, registers:

  ```
  typedef enum {is_field, is_key, is_instruc, is_label, is_reg, is_error} symTag;
  ```

- Naturally, our various entities are structures themselves. The big change we make is that *the first element of each such structure is a Symbol*, as we saw in the InstrucDef above.

This simplifies our code somewhat, and ensures that the tags are always present.

3.2. Initialisation

Now, we'll need to declare the important symbols for the assembler:

```
    // fill in the elements for fields
    for (i = 0; i < numFields; i ++) {
        // copy the FieldDef values from gField[i]
        gFieldDefs[i] = gInitFields[i];
        symInitSym(&gFieldDefs[i].sym, gFieldMapping[i].name);
        if (gFieldMapping[i].fieldtag != i) {
            printf("\nTag Error in fieldmapping %s",gFieldMapping[i].name);
            }
        symInstallSym(st, &gFieldDefs[i].sym);
        // mark the symbol as being for a field
        gFieldDefs[i].sym.tag = is_field;
        // set up the symbol
        if (gReport) printFieldDef(&gFieldDefs[i]);
    }

    // fill in the elements for instructions
    printf("\n\nSet up the instruction definitions...");
    for (i = 0; i < numInstrDefs; i++) {
        // initialise our symbol
        symInitSym(&gInstrucDefs[i].sym, gInstructionMnemonics[i].mnem);
        if (gReport) printSym(&gInstrucDefs[i].sym);
        // install our symbol in the symTab
        symInstallSym(st, &gInstrucDefs[i].sym);
        if (gReport) printSym(&gInstrucDefs[i].sym);
        // mark our symbol as being an instruction
        gInstrucDefs[i].sym.tag = is_instruc;
        // print the instruction definition
```

```
        if (gReport) printInstrucDef(&gInstrucDefs[i]);
    }

    // fill in the elements for keywords
    installKeyword(st, k_title, "title");
    installKeyword(st, k_start, "start");
    installKeyword(st, k_memory, "memory");
    installKeyword(st, k_three, "key_three");

    // fill in the elements for registers (r0..r15)
    for (i = 0; i < numRegs; i++) {
        char name[8];
        sprintf(name, "r%d", i);
        symInitSym(&gRegDefs[i].sym, name);
        symInstallSym(st, &gRegDefs[i].sym);
        gRegDefs[i].sym.tag = is_reg;
        gRegDefs[i].val = i;
        if (gReport) printRegDef(&gRegDefs[i]);
    }
```

Codepiece 22. Initialising the Symbols for the Assembler

To do this, we're using some pre-declared values. For example, in declaring the field names, we have some *#defines* (like *numFields*) and a couple of arrays. One is *gFieldDefs[],* which is an array of field definitions, each field being defined by

```
typedef struct  {
    Symbol sym;
    fieldSel n;
    fType ft;
    uint32 start;
    uint32 end;
    uint32 width;
    } FieldDef;
```

Codepiece 23. A field definition

And an array of useful values, held in *gInitFields[]*, which looks like this:

```
static FieldDef gInitFields[numFields] = {
    {{}, fmt_f, opcode, 15, 14, 0x3},
    {{}, op6_f, opcode, 13, 8, 0x3f},
    {{}, op2_f, opcode, 13, 12, 0x3},
    {{}, SL8_f, sval, 11, 4, 0xff},
    {{}, BD8_f, brdispl, 11, 4, 0xff},
    {{}, SL12_f, sval, 11, 0, 0xfff},
    {{}, BD12_f, brdispl, 11, 0, 0xfff},
    {{}, rd_f, regsel, 3, 0, 0xf},
    {{}, rs_f, regsel, 7, 4, 0xf},
    {{}, SL4_f, sval, 7, 4, 0xf}
    };
```

Codepiece 24. Encoding Information

We know exactly how many distinct fields there are in our architecture, so we can use nice simple arrays to hold the data. For the gInitFields[] array, we have initialisation values for each of the elements of a FieldDef except for the *Symbol.*

Our first step of initialisation is just to copy the structures from one array to the other:

```
        gFieldDefs[i] = gInitFields[i];
```

Since the initialisation doesn't include values for the symbol, we need to initialise that separately, thus:

```
        symInitSym(&gFieldDefs[i].sym, gFieldMapping[i].name);
```

In this, we make use of another array, *gFieldMappings[]*, which ties a name to a field selector:

```
static fieldMap gFieldMapping[] = {
    {fmt_f, "fmt"},
    {op6_f, "op6"},
    {op2_f, "op2"},
    {SL8_f, "SL8"},
```

```
    {BD8_f, "BD8"},
    {SL12_f, "SL12"},
    {BD12_f, "BD12"},
    {rd_f, "rd"},
    {rs_f, "rs"},
    {SL4_f, "SL4"},
    {maxField, "maxField"}
    };
```

Codepiece 25. Mapping field to field name

Then we install the symbol in our symbol table:

```
        symInstallSym(st, &gFieldDefs[i].sym);
```

And finally set the Symbol's tag value:

```
        gFieldDefs[i].sym.tag = is_field;
```

We use a similar approach to initialise instruction definitions and the installation of mnemonics into the symbol table. Creating the keywords is simpler.

Finally, we want to be able to write assembler instructions like *add r3, r5;* To do this we need to declare the names r0..r15 as registers. (In a real architecture, we might well need multiple register files, but for our simple example we'll limit all architectures to having just one register file). This is done pretty straightforwardly, if naughtily:

```
    for (i = 0; i < numRegs; i++) {
        char name[8];
        sprintf(name, "r%d", i);
        symInitSym(&gRegDefs[i].sym, name);
        symInstallSym(st, &gRegDefs[i].sym);
        gRegDefs[i].sym.tag = is_reg;
        gRegDefs[i].val = i;
        if (gReport) printRegDef(&gRegDefs[i]);
    }
```

Codepiece 26. Declaring the Register File

Why naughty? Because the name of the register file (more exactly, the names of the registers in the register file) should really be supplied by the header file, since it might well vary between architectures.

Once we've initialised the fields, the instructions, the keywords and the register file register names, we've captured all the information we need to parse and translate an assembler file. And, conveniently, the information we need is a bunch of declarations which in principle are generatable from the information the ADL compiler already has.

3.2.1. Instructions

Before parsing starts, we need to have something to capture the program. We'll use a Program structure:

```
typedef struct {
    char * title;
    uint32 placement_start;   // address at which the first instruction
                    // to be placed
    uint32 execution_start;   // address at which to start execution -
                    // which is at label main
    uint32 code_start;        // extent of executable program
    uint32 code_end;
    uint32 data_start;        // extent of data segment
    uint32 data_end;
    uint32 memory;        // needed size of memory
    Symbol * main;        // pointer to instruction which is first
                    // to be executed
    Queue instructions;       // queue of instructions
} Program;
```

Codepiece 27. The Program declaration

We keep track of sundry useful information, like where in memory the code is to be placed, where execution is to start, where any data might be placed, and -the instructions. We keep the instructions as a queue of *Instruction* structures. Before looking at these, we should recall that an instruction might be a branch to another instruction, and so we will want to compute a value for the branch displacement, and write it into the appropriate field in the instruction. This operation is often referred to as "fixing up" the instruction. To hold all the info we need, we create a *FixupInfo* structure and an *Instruction* structure:

```
typedef struct instruction Instruction;
```

```
typedef struct {
    FixupType fixupType;  // if it's an address reference or a
                          // displacement, we need to compute it
    fieldSel fixupField;  // the field definition for the fixup
    int32 displacement;   // the distance to a label in a branch
    Symbol * label_sym;   // the symbol corresponding to the label
                          // referenced
    Instruction * label;  // points at the Instruction representing
                          // the label
} FixupInfo;

struct instruction {
    qBlock header;
    uint32 address;          // the address of the instruction
    uint32 size;             // the size of the instruction in bytes.
                             // Labels have size 0
    instrucVal val;          // corresponds to mnemonic
    instrucType the_instruction;// the encoded instruction, of a type #defined in a generated header
    FixupInfo * fixupinfo;   // if there's a fixup to be done,
                             // this points at the relevant fixup info
};
```

Codepiece 28. Fixup information structures

An instruction can be a real instruction, or it can be a label. We will compute the address of each instruction so we can perform the necessary fixups. Labels are instructions of zero size; instructions have a type such as *uint16* or *uint32* (a type obviously implies/defines a size), and that type is *instrucType*, which is #defined in a header file (created by the adl compiler).

3.2.2. Parsing

To parse the asm program and hold the program representation, we re-use our tokeniser, queues and symbol table code. Parsing is straightforward:

```
// ----------------- parse -------------------

static Program * parse(const char * filename, symTab * st) {
    Program * program = newProgram();
    tokData * td = tokNewTokeniser((char *)filename);
    printf("\n-- Parsing '%s'...", filename) ;
    if (td == NULL) {
        printf("\nCannot create tokeniser. Done");
        return NULL;
    }
    Token t = t_none;

    while (t != t_eof) {
        t = tokGetToken(td);
        switch (t) {
            case t_eof:
                if (gReport) printf("\n== end of file ==\n");
                return program;

            case t_name:
            {
                // should be a mnemnonic
                Symbol * sym = symLookup(st, td -> tokPtr);
                if (sym) {
                    if (sym -> tag == is_instruc) {
                        // yay! we have a mnemonic
                        if (gReport) printf("\n\nmnemonic '%s'", sym -> name);
                        //tokReadToChars(td, "\n\r");
                        InstrucDef * instrucdef = (InstrucDef *)sym;
                                // an InstrucDef has its mnemonic's symbol embedded at the top
                        Instruction * instruc = createInstruc(program, instrucdef);
                        if (instruc) {
                            // goody...
```

```
                        if (gReport) printf("\n\t get fixed fields from definition..");
                        // find the fixed fields and their values
                        instruc -> the_instruction = fillInFixedFields(instruc, instrucdef);
                        // we have built the fixed part of the instruction up in encoding.
                        // Get the variable parts.
                        fillInOpndFields(td, st, instruc, instrucdef);

                    }
                    else {
                        printf("\nError in instruc def - couldn't find def for '%s'", sym -> name);
                        gInternalErrors++;
                    }

                }
                else if (sym -> tag == is_key) {
                    // ooo a keyword
                    if (gReport) printf("\nkeyword '%s'", sym -> name);
                    // what keyword?
                    KeyDef * keydef = (KeyDef *)sym;
                    switch (keydef -> key) {
                        case k_title:
                            // read and store the title
                            program -> title = getName(td, "program title");
                            break;
                        case k_memory:
                            // read the amount of memory we want for this program
                            program -> memory = getInt(td, "memory size");
                            break;

                        case k_start:
                            // read the hex start address,a placement command,saying where the
                            // first instruction is placed. It's NOT where execution starts
                            program -> placement_start = getInt(td, "start address");
                            break;

                        default:
                            tokReadToChars(td, "\n\r");
                    }
                }
                else if (sym -> tag == is_reg) {
                    printf("\nError - Found a register name outside an instructiondecode!(sym = %s,
tag = %d)", sym -> name, sym -> tag);
                    gParseErrors++;
                }
                else {
                    printf("\nalphanumeric token that's of an unrecognised sort (sym = %s, tag =
%d)", sym -> name, sym -> tag);
                    gParseErrors++;
                    tokReadToChars(td, "\n\r");
                }
            }
        }
            break;

        case t_larray:
            {
            // should be a label - [name]
            t = tokGetToken(td);
            if (t == t_name) {
                // goody. a label
                if (gReport) printf("\n== looks like we have a label '%s'..", td -> tokPtr);
                Symbol * sym = symLookup(st, td -> tokPtr);
                if (sym) {
                    if (sym -> tag == is_label) {
                        // Uh uh! we have an already-declared label [illegal]
```

```
                                printf("\nError - duplicate label '%s'", sym -> name);
                                gParseErrors++;
                                tokReadToChars(td, "\n\r");
                            }
                            else if (sym -> tag == is_fwdlabel) {
                                if (gReport) printf("\n\tfound a forward label!");
                                Label * label = (Label *) sym;
                                createLabelInstruc(program, sym, label);
                                tokGetThisToken(td, t_rarray);
                            }
                            else {
                                printf("\nError - unexpected token '%s' - expected new (or fwd reference)
label name", sym -> name);
                                gParseErrors++;
                            }
                        }
                        else {
                            // goody a new label - create stuff
                            Label * label = newLabel(td -> tokPtr);
                            Symbol * sym = (&label -> sym);
                            // install it
                            symInstallSym(st, sym);
                            sym -> tag = is_label;
                            // if the label is 'main' note we have a start address
                            if (strcmp(td -> tokPtr, "main") == 0) {
                                if (gReport) printf("\n-- Found label main in source..");
                                program -> main = sym;
                                label -> is_main = TRUE;
                            }
                            else {
                                label -> is_main = FALSE;
                            }
                            // we create a dummy instruction of program size zero
                            createLabelInstruc(program, (Symbol *)label, (Label *)sym);
                            tokReadToChars(td, "\n\r");
                            }

                        }
                    }
                    break;
                default:
                    printf("\nunrecognised token '%s'", td -> tokPtr);
                    gParseErrors++;
                    break;

                }
        }
        return program;
    }
```

Codepiece 29. The parsing function

The parsing code is, as noted, straightforward, although this version suffers badly from poor error handling. The parser returns a *Program* structure, which must then be fixed up and then output as a loadable program.

3.2.3. Fixing Up the Program

Now we need to fixup up the program. This is also straightforward (at least, for a fixed-length instruction architecture). The first thing to do is to run through all the instructions, giving each one an address, computed starting from the captured information saying where the code segment starts, and computing the address o the next instruction as "address of this one plus the instruction size". The one piece worth noting is we check each instruction to see if it's a label named *main*; if it is, we've found the address at which to start execution. We note where to start, and change the tag of the label to be an ordinary label:

```
    static void fixupProgram(Program * program) {
        Instruction * instruc = (Instruction *)qFirstBlock(&program -> instructions);
```

```
uint32 addr = program -> placement_start;
printf("\n\nFixing up...");
// 1. give every instruction an address
printf("\n\t..giving instructions address and finding main()...");
program -> code_start = program -> placement_start;
while (instruc) {
     instruc -> address = addr;
     addr += instruc -> size;
     FixupInfo * fixupinfo = instruc -> fixupinfo;
     if (fixupinfo && (fixupinfo -> fixupType) == is_main) {
          // capture this address as start address
          if (gReport) printf("\n==found main in instructions.");
          program -> execution_start = addr;
          // and change its fixup to is_a_label
          fixupinfo -> fixupType = is_a_label;
     }
     instruc = qNext(instruc);
}
// 2. find all labels and update their symbols to point at the instructions
printf("\n\tfinding all labels and pointing their symbols at the relevant instructions...");
instruc = (Instruction *)qFirstBlock(&program -> instructions);
while (instruc) {
     FixupInfo * fixupinfo = instruc -> fixupinfo;
     if (fixupinfo && (fixupinfo -> fixupType == is_a_label)) {
          // find its sym and point the sym at the instruc
          Symbol * sym = fixupinfo -> label_sym;
     }
     instruc = qNext(instruc);
}

// 3. specify top code segment
printf("\n\tcreating segments...");
program -> code_end = addr;


// 4. is there a data segment specified?
// if not let the data seg be the space above the code
if ((program -> data_start == 0) && (program -> data_end == 0)) {
     // compute an address above the top of code
     addr = addr + 4;
     addr = addr & (~0xf);
     program -> data_start = addr;
     program -> data_end = program -> memory - 4;
}

// 5. do the instruction fixups
printf("\n\tfix up branches etc...");
instruc = (Instruction *)qFirstBlock(&program -> instructions);
while (instruc) {
     FixupInfo * fixupinfo = instruc -> fixupinfo;
     if (fixupinfo) {
     switch (fixupinfo -> fixupType) {
          case no_fixup:
          case is_a_label:
          case is_main:
               break;
          case displacement_fixup:
               // compute distance between this instruction and the
               // mentioned label
          {
               // my label_sym points at the destination instruction
               Instruction * dest = (Instruction *) fixupinfo ->
                    label_sym;
               // compute the distance
               printf("\nFixup: instruc at %d label at %d",
```

```
                    instruc -> address, dest -> address);
                int32 d = dest -> address - instruc -> address;
                printf("\t -> displacement of %d [0x%08x] bytes", d, d);
                // find the fixup field
                fieldSel f = fixupinfo -> fixupField;
                FieldDef * field = &gFieldDefs[f];
                // manipulate the d appropriately
                d = d & field -> width;
                d = d << field -> end;
                printf("\n\tFixup field: '%s': width 0x%08x, end %d",
                    field -> sym.name, field -> width, field -> end);
                // write into the instruction encoding
                instruc -> the_instruction |= d;
                printf("\n\tinstruction now 0x%4x",
                    instruc -> the_instruction);

            }
                break;
            case absolute_fixup:
                // insert the actual address of mentioned label
                break;
            }
        }
        instruc = qNext(instruc);
    }
}
```

Codepiece 30. Fixing up the instructions

3.3. Putting it all together

We now know what the assembler has to contain, and how it does it; it remains to teach the ADL compiler how to do this. In generating the model, we created just a model.c and a model.h file. Here, we will need to create two header files and an executable file.

First, recall that these generated files are to be compiled with other 'fixed' files. One such fixed file is a file containing declarations common to all our generated assemblers; that file is *asmDefinitions.h*:

```
//
//  asmDefinitions.h
//  asm
//
//  Created by Pete on 5/2/17.
//  Copyright © 2017 Kiva Design Groupe LLC. All rights reserved.
//

#ifndef fields_h
#define fields_h

// data types for the assembler

// a Program
typedef struct {
    char * title;
    uint32 placement_start;       // address at which the first instruction is placed
    uint32 execution_start;       // address at which to start execution - which is at label main
    uint32 code_start;            // extent of executable program
    uint32 code_end;
    uint32 data_start;            // extent of data segment
    uint32 data_end;
    uint32 memory;                // needed size of memory
    Symbol * main;                // pointer to instruction which is first to be executed
    Queue instructions;           // queue of instructions
} Program;
// the types of fields. These should be an ADL invariant, but generating makes things more explicit type-
wise
typedef enum {opval, uval, sval, brdispl, regsel, absaddr} fType;
```

```
/*
 Fixups:
 no_fixup - the instruction's fields don't depend on anything else
 displacement_fixup - for branches, which are relative. the BDx field needs computing. So build all the
instructions and then run back through them putting in the
 absolute_fixup - we need an actual address
 is_a_label - this is a label and so has an address but no size
 is_main - this is a label for the executable entry point
 */
typedef enum {no_fixup, displacement_fixup, absolute_fixup, is_main, is_a_label} FixupType;

// the actual program instructions. These are queued up on the Program structure
typedef struct instruction Instruction;

typedef struct {
    FixupType fixupType;      // if it's an address reference or a displacement, we need to compute it
    fieldSel fixupField;      // the field definition for the fixup
    int32 displacement;           // the distance to a label in a branch
    Symbol * label_sym;           // the symbol corresponding to the label referenced
    Instruction * label;      // points at the Instruction representing the label
} FixupInfo;

struct instruction {
    qBlock header;
    uint32 address;               // the address of the instruction
    uint32 size;              // the size of the instruction in bytes. Labels have size 0
    instrucVal val;               // corresponds to mnemonic
    instrucType the_instruction;// the encoded instruction
    FixupInfo * fixupinfo;        // if there's a fixup to be done, this points at the fixup info
};

// an instruction definition - parsing information and encoding information.
typedef struct {
    Symbol sym;
    instrucVal val;
    uint32 numOpndFields;             // the number of fields in the assembler syntax
    fieldSel OpndFields[maxFields];   // the fields in the assembler syntax (like rd and rs)
    uint32 numFixedFields;            // the number of fixed fields in the encoded instruction
    fieldSel fixedFields[maxFields];  // the actual fixed fields in the encoded instruction(like fmt, op6)
    uint32 fieldValues[maxFields];    // the values for those fields
    } InstrucDef;

typedef enum {k_title, k_start, k_memory, k_three, k_max} Key;

// a Key

typedef struct {
    Symbol sym;
    Key key;
} KeyDef;

// a Label

typedef struct {
    Symbol sym;
    uint32 is_main;                     // TRUE if this is 'main'
    Instruction * my_instruc;         // points at the instruction representing the label
} Label;

//a Register

typedef struct {
    Symbol sym;
    char name[16];
```

```
    uint32 val;
} RegDef;

// now, an actual field definition structure
typedef struct  {
    Symbol sym;
    fieldSel n;
    fType ft;
    uint32 start;
    uint32 end;
    uint32 width;
} FieldDef;

// functions in main callable by generated asm.c

void handleRegSelField(symTab *st, tokData * td, fieldSel field, Instruction * instruc);
void handleDisplacementField(symTab *st, tokData * td, fieldSel field, Instruction * instruc);
void handleConstantField(symTab *st, Token t, tokData * td, fieldSel field, Instruction * instruc);
void handleFixedField(void);

void parseErr(tokData *td, char * msg);
int doReport(void);

// in generated asm.c
void getField(symTab * st, tokData * td, fieldSel field, FieldDef * f, Instruction * instruc);

#endif /* fields_h */
```

Codepiece 31. The common asmDefinitions.h file.

Note that it needs some information from the generated assembler, such as *instrucType*. We must therefore generate a header file to be #included before asmDefinitions.h which is asmFields.h:

```
//
// asmFields.h
//

/*

CAUTION: This file generated by:
simpleADL [simpleADL 1.0v20]
    using tokeniser package simpleTokeniser 1.0v9 [Jan 2017]
    using queue package simpleQueues 1.0v0 [Jan 16 2017]
    using symbol table simpleSymTab [January 2017] 1.0v0
        All software copyright Kiva Design Groupe LLC 2017. All rights reserved. See license for terms of
use

Do not edit and expect the changes to stick!
*/
#define instrucType uint16
#define instrucBytes (2)
#define numInstrDefs  (23)
#define numKeys       (4)
#define numFields     (10)
#define numRegs       (16)
#define maxFields     (4)

// declare the field selectors, one for each defined field.The selector is just the name of the field
suffixed by '_f'
typedef enum {fmt_f, op6_f, op2_f, SL8_f, BD8_f, SL12_f, BD12_f, rd_f, rs_f, SL4_f, maxField} fieldSel;

// tags for all the instruction mnemonics
typedef enum {i_add,
    i_sub, i_mul, i_div, i_rem, i_cpy, i_ld8, i_ld16, i_ld32, i_halt, i_bne, i_ldc8, i_beq, i_jmp, i_ldc4,
i_addc, i_subc,
    i_mulc, i_divc, i_remc, i_ld8c, i_ld16c, i_ld32c
```

```
    } instrucVal;
```

Codepiece 32. The generated asmFields.h file

And there's a second header file, asm.h, which contains the rest of the generated declarations:

```
//
// asm.h
//

/*

CAUTION: This file generated by:
simpleADL [simpleADL 1.0v20]
    using tokeniser package simpleTokeniser 1.0v9 [Jan 2017]
    using queue package simpleQueues 1.0v0 [Jan 16 2017]
    using symbol table simpleSymTab [January 2017] 1.0v0
        All software copyright Kiva Design Groupe LLC 2017. All rights reserved. See license for terms of
use

Do not edit and expect the changes to stick!
*/
// asm.h: header file for 'archTest'
#ifdef isAsmMain
char * archfolder = "/Volumes/Oxford Road/Users/pete/Dropbox/SoftwareProjects-Working/ArchProj/archModels/
archTest";
#endif


// a mapping between field name and fieldName
typedef struct {
    fieldSel fieldtag;
    char * name;
    } fieldMap;

static fieldMap gFieldMapping[] = {
    {fmt_f, "fmt"},
    {op6_f, "op6"},
    {op2_f, "op2"},
    {SL8_f, "SL8"},
    {BD8_f, "BD8"},
    {SL12_f, "SL12"},
    {BD12_f, "BD12"},
    {rd_f, "rd"},
    {rs_f, "rs"},
    {SL4_f, "SL4"},
    {maxField, "maxField"}
    };

// initalisations for the field definitions
static FieldDef gInitFields[numFields] = {
    {{}, fmt_f, uval, 15, 14, 0x3},
    {{}, op6_f, uval, 13, 8, 0x3f},
    {{}, op2_f, uval, 13, 12, 0x3},
    {{}, SL8_f, sval, 11, 4, 0xff},
    {{}, BD8_f, brdispl, 11, 4, 0xff},
    {{}, SL12_f, sval, 11, 0, 0xfff},
    {{}, BD12_f, brdispl, 11, 0, 0xfff},
    {{}, rd_f, regsel, 3, 0, 0xf},
    {{}, rs_f, regsel, 7, 4, 0xf},
    {{}, SL4_f, sval, 7, 4, 0xf}
    };

// map the mnemonics to the instruction 'values'
typedef struct {
    char * mnem;
    instrucVal mnemVal;
```

```
    } iMap;

static iMap gInstructionMnemonics[] = {
    {"add", i_add},
    {"sub", i_sub},
    {"mul", i_mul},
    {"div", i_div},
    {"rem", i_rem},
    {"cpy", i_cpy},
    {"ld8", i_ld8},
    {"ld16", i_ld16},
    {"ld32", i_ld32},
    {"halt", i_halt},
    {"bne", i_bne},
    {"ldc8", i_ldc8},
    {"beq", i_beq},
    {"jmp", i_jmp},
    {"ldc4", i_ldc4},
    {"addc", i_addc},
    {"subc", i_subc},
    {"mulc", i_mulc},
    {"divc", i_divc},
    {"remc", i_remc},
    {"ld8c", i_ld8c},
    {"ld16c", i_ld16c},
    {"ld32c", i_ld32c}
    };


// The list of instruction definitions
static InstrucDef gInstrucDefs[numInstrDefs] = {
    {{}, i_add, 2, {rd_f, rs_f}, 2, {fmt_f, op6_f}, {0, 1}},
    {{}, i_sub, 2, {rd_f, rs_f}, 2, {fmt_f, op6_f}, {0, 2}},
    {{}, i_mul, 2, {rd_f, rs_f}, 2, {fmt_f, op6_f}, {0, 3}},
    {{}, i_div, 2, {rd_f, rs_f}, 2, {fmt_f, op6_f}, {0, 4}},
    {{}, i_rem, 2, {rd_f, rs_f}, 2, {fmt_f, op6_f}, {0, 5}},
    {{}, i_cpy, 2, {rd_f, rs_f}, 2, {fmt_f, op6_f}, {0, 6}},
    {{}, i_ld8, 2, {rd_f, rs_f}, 2, {fmt_f, op6_f}, {0, 60}},
    {{}, i_ld16, 2, {rd_f, rs_f}, 2, {fmt_f, op6_f}, {0, 61}},
    {{}, i_ld32, 2, {rd_f, rs_f}, 2, {fmt_f, op6_f}, {0, 62}},
    {{}, i_halt, 0, {}, 2, {fmt_f, op6_f}, {0, 63}},
    {{}, i_bne, 2, {rd_f, BD8_f}, 2, {fmt_f, op2_f}, {1, 0}},
    {{}, i_ldc8, 2, {rd_f, SL8_f}, 2, {fmt_f, op2_f}, {1, 1}},
    {{}, i_beq, 2, {rd_f, BD8_f}, 2, {fmt_f, op2_f}, {1, 2}},
    {{}, i_jmp, 1, {BD12_f}, 2, {fmt_f, op2_f}, {2, 0}},
    {{}, i_ldc4, 2, {rd_f, SL4_f}, 2, {fmt_f, op6_f}, {3, 0}},
    {{}, i_addc, 2, {rd_f, SL4_f}, 2, {fmt_f, op6_f}, {3, 1}},
    {{}, i_subc, 2, {rd_f, SL4_f}, 2, {fmt_f, op6_f}, {3, 2}},
    {{}, i_mulc, 2, {rd_f, SL4_f}, 2, {fmt_f, op6_f}, {3, 3}},
    {{}, i_divc, 2, {rd_f, SL4_f}, 2, {fmt_f, op6_f}, {3, 4}},
    {{}, i_remc, 2, {rd_f, SL4_f}, 2, {fmt_f, op6_f}, {3, 5}},
    {{}, i_ld8c, 2, {rd_f, SL4_f}, 2, {fmt_f, op6_f}, {3, 60}},
    {{}, i_ld16c, 2, {rd_f, SL4_f}, 2, {fmt_f, op6_f}, {3, 61}},
    {{}, i_ld32c, 2, {rd_f, SL4_f}, 2, {fmt_f, op6_f}, {3, 62}}
    };
```

Codepiece 33. The generated asm.h file

When parsing the assembler source, the assembler needs to know the format of each instruction; since each architecture has its own names for operands and opcode fields, this is necessarily per-architecture and must be created to match the architecture. This is simply done by generating asm.c and providing therein a function to do the right thing; the function just provides a mapping between a field name and the type of field it is, calling an appropriate routine common to all of our assemblers :

```
//
// asm.c
//
```

```
/*

CAUTION: This file generated by:
simpleADL [simpleADL 1.0v20]
    using tokeniser package simpleTokeniser 1.0v9 [Jan 2017]
    using queue package simpleQueues 1.0v0 [Jan 16 2017]
    using symbol table simpleSymTab [January 2017] 1.0v0
        All software copyright Kiva Design Groupe LLC 2017. All rights reserved. See license for terms of
use

Do not edit and expect the changes to stick!
*/
// asm.c: implementation file for 'archTest'
#include <stdio.h>
#include "types.h"
#include "utilities.h"
#include "Tokens.h"
#include "queues.h"
#include "symbol.h"
#include "asmFields.h"
#include "asmDefinitions.h"
#include "asm.h"


// --------------- getField -----------------

void getField(symTab * st, tokData * td, fieldSel field, FieldDef * f, Instruction * instruc) {
// get the field demanded by the instruction format
// get the next significant token, and process it according to the field definition
Token t = tokGetToken(td);
switch (field) {
    case fmt_f:
        handleConstantField(st, t, td, field, instruc);
        break;

    case op6_f:
        handleConstantField(st, t, td, field, instruc);
        break;

    case op2_f:
        handleConstantField(st, t, td, field, instruc);
        break;

    case SL8_f:
        handleConstantField(st, t, td, field, instruc);
        break;

    case BD8_f:
        handleDisplacementField(st, td, field, instruc);
        break;

    case SL12_f:
        handleConstantField(st, t, td, field, instruc);
        break;

    case BD12_f:
        handleDisplacementField(st, td, field, instruc);
        break;

    case rd_f:
        handleRegSelField(st, td, field, instruc);
        break;

    case rs_f:
```

```
            handleRegSelField(st, td, field, instruc);
            break;

        case SL4_f:
            handleConstantField(st, t, td, field, instruc);
            break;

        case maxField:
            parseErr(td, "\nError - maxField is an illegal field selector value.");
            break;

        }
    }
```

Codepiece 34. The generated asm.c file