The M1 Processor Circuit for Sigma16

John T. O'Donnell

October 2021

Contents

1 Introduction

Sigma16 is a computer architecture designed for research and teaching in computer systems. M1 is a CPU: it is a digital circuit that executes the Core subset of Sigma16.

M1 is a synchronous circuit designed at the level of logic gates and flip flops. It is specified using the Hydra hardware description language, which describes precisely the components and how they are connected, and which can simulate the circuit.

The Circuit directory contains the M1 circuit, which is organized as several subsystems. The Run module defines a simulation driver; this is a command line text interface that provides human readable inputs and outputs. The ReadObj module is used to boot the system: it reads an object code file produced by the Sigma16 assembler and automatically generates the inputs required to make the circuit boot the program.

2 Sigma16 Core

First instruction word (RRR and RX)

op	d	a	b

The fields are:

field	bit index	size	description
op	15–12	4 bits	operation code
d	11–8	4 bits	destination register address
a	7–4	4 bits	source register address a
b	3–0	4 bits	source register address b

Second instruction word (RX only)

field	bit index	size	description
disp	15–0	16 bits	displacement

There is only one field, the displacement (sometimes abbreviated to disp)

Mnemonic	ISA	P	Fmt	Args	Code	Effect
add	Core		RRR	RRR	0	Rd := Ra + Rb
sub	Core		RRR	RRR	1	Rd := Ra - Rb
mul	Core		RRR	RRR	2	Rd := Ra * Rb
div	Core		RRR	RRR	3	Rd := Ra / Rb, R15 := Ra rem Rb
cmp	Core		RRR	RR	4	R15 := Ra cmp Rb
trap	Core		RRR	RRR	b	user interrupt
lea	Core		RX	RX	f,0	Rd := ea
load	Core		RX	RX	f,1	Rd := M[ea]
store	Core		RX	RX	f,2	M[ea] := Rd
jump	Core		RX	RX	f,3	pc := ea
jumpc0	Core		RX	kX	f,4	if R15.k=0 then pc := ea
jumpc1	Core		RX	kX	f,5	if R15.k=1 then pc := ea
jal	Core		RX	RX	f,6	Rd := pc, pc := ea

3 Running the system

3.1 Installation

- Install ghc from www.haskell.org

- The current version of M1 contains Hydra in a subdirectory, so you don't need to install it separately. However, it may be necessary to install a few Haskell packages, using the following commands:

```
cabal update
cabal install mtl
cabal install parsec
cabal install ansi-terminal
```

3.2 Quick start

The following commands will start M1, boot an example program, and run it to completion. The object code must be in Sigma16 object format, which is produced by the Sigma16 assembler. Assembling source program Add.asm.txt will produce the machine language code in Add.obj.txt.

```
$ cd M1
$ ghci
ghci> :load Run
ghci> :main programs/Add
M1> run
M1> quit
ghci> :quit
```

After you have started a program with `:main programs/Add`, you can run one clock cycle simply by pressing Enter instead of typing run. Do this repeatedly to see how the circuit evolves through a sequence of clock cycles.

If the object code is n words long, it will take the circuit n clock cycles to boot it. You can reach the point when the circuit actually starts running the program by entering these commands:

```
ghci> :main programs/Add
M1> break reset
M1> run
```

Here is a brief example of Using breakpoints and register/memory display. The commands are described in more detail below.

```
$ ghci
ghci> :load Run
ghci> main programs/ArrayMax
```

```
M1> break reset    set breakpoint
M1> run            run until reset=1
M1> (enter)        run just one  clock cycle
M1 run
M1> regs           print the contents of the register file
M1> mem 0 30       print memory from address 0 to 30
```

3.3 Configuration

There is a directory `circuits/programs` which contains several Sigma16 programs. The examples below use these programs, and you can put your own programs there too. If you do so, you can skip this section.

However, you may wish to keep the circuits directory and your Sigma16 programs directory in different locations in your file system. If you do that, you may end up with long file paths to your object code, and this can make it hard to type in the command to load a program. The M1 driver has a facility to make that easier.

If the file `circuits/fileprefix.txt` exists, then the text in its first line is attached to the beginning of the file specified in the argument to the `:main` command. For example, suppose the full path to your object code file is `/some/long/path/to/your/code/MyProgram.obj.txt`. Then you can put `/some/long/path/to/your/code/` into `fileprefix.txt`, and then run the program with just `:main MyProgram`.

If you don't want anything prefixed to your file arguments, either set fileprefix.txt to a blank line, or simply delete it.

3.4 An example program

Before examining the circuit, let's begin by taking a Sigma16 assembly language program and running it on the circuit. This is `circuits/programs/Add.asm.txt`:

```
; Add: a minimal program that adds two integer variables
; This file is part of Sigma16.   https://jtod.github.io/home/Sigma16

; Execution starts at location 0, where the first instruction will be
; placed when the program is executed.

      load   R1,x[R0]    ; R1 := x
      load   R2,y[R0]    ; R2 := y
      add    R3,R1,R2    ; R3 := x + y
      store  R3,z[R0]    ; z := x + y
      trap   R0,R0,R0   ; terminate

; Expected result: z = x + y = 23 + 14 = 37 (hex 0025)

; Static variables are placed in memory after the program

x     data  23
y     data  14
z     data   0
```

3.5 Assemble the program

Computers don't run assembly language, they run machine language. So we need to translate the program. There are several ways to do this:

- *Command.* If you have installed the Sigma16 tools on youc computer, enter this command: `sigma16 assemble programs/Add`
- *Sigma16 app.* Launch Sigma16, load the example program, assemble it, and save the object code in a file. To see the object code, click the Object Code link in the Assembler page. Then copy and paste the text into a text editor and save it. The file must be named `Add.obj.txt`
- *Assemble it by hand.* It's important to know *how* to assemble a program by hand, and it's worth doing one or two times. But once you understand how to translate from assembly to machine language, it's better to use the software tools. Hand assembly is particularly useful when experimenting with new instructions in an architecture.

3.6 Run the machine code on the M1 circuit

Now you can run the machine language program `Add.obj.txt` on the circuit:

- `$ cd M1` The ghci command must be executed in this directory.
- `$ ghci` This launchs Haskell.
- `ghci> :load Run` Launch Hydra and M1.
- `ghci> :main programs/Add` Load the machine language program and prepare to boot it.
- `M1> run` Boot the program and run it on the circuit.

- `M1> quit` Leave M1 driver, return to ghci.
- `ghci> :quit` Leave ghci, return to the shell.

3.7 Breakpoints

The machine may execute many clock cycles before it reaches a state that you're interested in. For example, if you want to examine exactly how the circuit executes a jal instruction, you need to get through the boot process and then all the instructions that execute before the jal. This can take a long time, and you may have to do it repeatedly.

The M1 simulation driver provides *breakpoints* which alleviate this problem. The idea is that you specify that a bit signal of interest is a breakpoint. Then when you enter a run command, it will perform clock cycles repeatedly until the breakpoint signal becomes 1. At that point the simulation stops and you can examine the machine state in detail, and single step (or run) from that point on.

The `help` command gives you a list of signals that are registered so they can be used as a breakpoint.

One useful breakpoint signal is `reset`. When you start the system it may take a considerable number of clock cycles to boot the machine language program. You can skip over those cycles and go directly to the point where the machine starts executing the program with the following commands. This will run the simulation without stopping, until the reset signal becomes 1, and then it will stop. That way you can start single stepping through the program, but don't have to single step through the boot.

```
break reset
run
```

Another useful technique is to go quickly to the point where the machine is starting to execuate a particular instruction that you're interested in. The convention is that the first state of the control algorithm for an instruction is named `st_instr0`. For example, if you want to watch in detail how the circuit executes a load instruction, use these commands:

```
break st_load0
run
```

3.8 Dumping the register file and memory

The simulation driver shows the values of all the output signals from the circuit, and this includes key registers, such as `pc`, `ir`, and `adr`. However, most of the computer's state is in the register file and the memory, and these are not directly visible.

If you follow all the details of every clock cycle, you can work out the contents of the register file and the memory. But this may be impractical. If you want to know what is in memory at some particular address, there is no bound on how far back you would have to search to find the point when something was stored in that location.

The simulation driver provides two commands that solve this problem. The `regs` command prints the contents of the register file:

```
regs
```

The `mem` command prints the contents of memory from a starting address to an ending address. The following command prints memory up to location 20:

```
mem 0 20
```

These commands are not implemented by looking into the simulator's internal data structures. Indeed, the simulator doesn't know anything about the circuit apart from the signal values. The commands are implemented by the Input/Output system, using direct memory access (DMA) and cycle stealing. This is the way testing is done on real hardware. You can see that the dump commands require a number of clock cycles to perform, even though the driver doesn't show all the internal signals during those cycles.

3.9 Commands

Prompts

```
$          is the bash shell prompt
ghci>      is the ghci prompt
M1>        is the simulation driver prompt
```

Useful ghci commands (see ghc User Guide for full documentationO

```
:r          reload after editing any of the code
uparrow     repeat previous command
:q          exit ghci, go back to shell
^C          stop and return to ghci prompt
```

Simulation driver commands: enter help for a list.

4 Interface

The M1 system contains several subsystems (Datapath, Control, etc.). Each of these subsystems takes several inputs and produces several outputs. Those inputs and outputs are collected together into records defined in the Interface module. The actual connections between the subsystems are then defined in the `System` module.

4.1 Records with named fields

When a circuit has a small number of inputs or outputs, it's straightforward to provide them in a fixed order. For example, here is the definition of a multiplexer:

```
mux1 :: Signal 1 => a -> a -> a -> a
mux1 c x y = or2 (and2 (inv c) x)
                 (and2 c y)
```

To use it, you need to know that the first input `c` is the control, that the input `x` is output if `c` is zero, and the input `y` is output if `c` is one.

However, complex circuits may have too many inputs and outputs to be able to remember all their positions in a list of ports. Even if you remember that the signal you want is the 19th one, it's awkward to get access to it. Some modern chips have thousands of input and output signals.

The solution is to identify the signals by name and to collect a group of named signals in a *record*. Here is a definition from the Datapath module, which provides a number of output signals that are collected into a record with type `DPoutputs a`:

```
data DPoutputs a = DPoutputs
     { aluOutputs :: ([a], [a])
     , r :: [a]        -- alu output
     , ccnew :: [a]     -- alu output
     , ma :: [a]
...
     }
```

The Datapath module defines the signals: it contains an equation for each element of the record giving the value.

```
...
aluOutputs = ...
r = ...
ccnew = ...
...
```

The record itself is named `dp` and is defined by this equation:

```
dp = DPoutputs {..}
```

The notation `{..}` means that each defined value whose name is listed in the record type (`DPoutputs`) should be included in the actual record (`dp`). For example, the record type `DPoutputs a` defines a field named `ccnew` and there is an equation defining the value `ccnew`, so `ccnew` is included in the record `dp`.

The definition of the datapath circuit is an equation where the right hand side says that the output is `dp`, which is defined to be the record of signals `dp`:

```
datapath (CtlSig {..}) (SysIO {..}) memdat = dp
  where

-- Interface
    dp = DPoutputs {..}
```

The first input to the datapath circuit is `(CtlSig {..})`, which is a record of signals output by the control circuit. When used as an input, this `{..}` notation means that every field in the `CtlSig` record defines the corresponding name. With this notation, you don't need to write a separate equation for every element of the `CtlSig` record.

The `{..}` notation makes it easy to add a new signal to a record. For example, suppose you modify the datapath to contain two new signals: `pqr` which is a bit, and `xyz` which is a word of bits. The following changes are required:

- Add equations defining the new signals to the `Datapath` module:

```
pqr = ...
xyz = ...
```

- Add the new signals to the record definition in the `Interface` module:

```
data DPoutputs a = DPoutputs
  { ...
  , pqr :: a
  , xyz :: [a]
  ...
}
```

- Now you can use `pqr` and `xyz` in any module that receives the `DPoutputs`.

4.2 The Interface module

```
-- Sigma16 M1: Interface.hs
-- John T. O'Donnell, 2021
-- See Sigma16/README and https://jtod.github.io/home/Sigma16/

{-# LANGUAGE NamedFieldPuns #-}

module Circuit.Interface where
import HDL.Hydra.Core.Lib

------------------------------------------------------------------------
-- Input/Output
------------------------------------------------------------------------

data SysIO a = SysIO
  { io_DMA        :: a
  , io_memStore   :: a
  , io_memFetch   :: a
  , io_regFetch   :: a
  , io_address    :: [a]
  , io_data       :: [a]
  }

------------------------------------------------------------------------
-- Control signals
------------------------------------------------------------------------

{- The behaviour of the datapath is determined by the values of the
control signals.  A naming convention is used for these signals.  Each
control signal has a name with three parts, separated by _.  The first
```

```
part is ctl (to indicate that it's a control signal).  The second part
is the name of the subsystem in the datapath which is being
controlled, and the third part is the specific operation being
commanded by the signal. -}

data CtlSig a = CtlSig
  {
-- Controls for system
    cpu,        -- indicates dff for state generates state controls
    condcc,

-- Controls for ALU
   ctl_alu_a,   -- 3-bit alu operation code
   ctl_alu_b,   --   "
   ctl_alu_c,   --   "
   ctl_x_pc,    -- Transmit pc on x (if 0, transmit reg[sa])
   ctl_y_ad,    -- Transmit ad on y (if 0, transmit reg[sb])

-- Controls for register file
   ctl_rf_ld,   -- Load  register file (if 0, remain unchanged)
   ctl_rf_ldcc, -- Load  R15 (if 0, remain unchanged; ld takes priority)
   ctl_rf_pc,   -- Input to register file is pc (if 0, check ctl_rf_alu)
   ctl_rf_alu,  -- Input to register file is ALU output r (if 0, use m)
   ctl_rf_sd,   -- Use ir_d as source a address (if 0, use ir_sa)

-- Controls for system registers
   ctl_ir_ld,   -- Load ir register (if 0, remain unchanged)
   ctl_pc_ld,   -- Load pc register (if 0, remain unchanged)
   ctl_pc_ad,   -- Input to pc is ad (if 0, r)
   ctl_ad_ld,   -- Load ad register (if 0, remain unchanged)
   ctl_ad_alu,  -- Obtain ad input from alu (if 0, from memory data input)

-- Controls for memory
   ctl_ma_pc,   -- Transmit pc on memory address bus (if 0, transmit addr)
   ctl_sto      -- Memory store (if 0, fetch)
   :: a         -- all controls are bit signals
  }

data CtlState a = CtlState
  {dff_instr_fet, st_instr_fet,
   dff_dispatch, st_dispatch,
   dff_add, st_add,
   dff_sub, st_sub,
   dff_mul0, st_mul0,
   dff_div0, st_div0,
   dff_cmp, st_cmp,
   dff_trap0, st_trap0,
   dff_lea0, st_lea0, dff_lea1, st_lea1, dff_lea2, st_lea2,
   dff_load0, st_load0, dff_load1, st_load1, dff_load2, st_load2,
   dff_store0, st_store0, dff_store1, st_store1, dff_store2, st_store2,
   dff_jump0, st_jump0, dff_jump1, st_jump1, dff_jump2, st_jump2,
   dff_jumpc00, st_jumpc00, dff_jumpc01, st_jumpc01, dff_jumpc02, st_jumpc02,
   dff_jumpc10, st_jumpc10, dff_jumpc11, st_jumpc11, dff_jumpc12, st_jumpc12,
   dff_jal0, st_jal0, dff_jal1, st_jal1, dff_jal2, st_jal2
   :: a         -- all control states are bit signals
  }

data DPoutputs a = DPoutputs
     { aluOutputs :: ([a], [a])
     , r :: [a]        -- alu output
     , ccnew :: [a]     -- alu output
     , ma :: [a]
     , md :: [a]
     , a :: [a]
     , b :: [a]
     , cc :: [a]
     , ir :: [a]
     , ir_op :: [a]
     , ir_d :: [a]
     , ir_sa :: [a]
     , ir_sb :: [a]
     , pc :: [a]
     , ad :: [a]
     , x :: [a]
     , y :: [a]
     , p :: [a]
     , q :: [a]
     }
```

5 Datapath

The datapath of a processor contains the registers, the circuits that perform calculations (ALU and functional units), and the buses that connect them. All of these subsystems take control inputs that determine their behavior. Those control signals are generated by the control system, which is not part of the datapath.

5.1 ALU: the Arithmetic and Logic Unit

The ALU (arithmetic and logic unit) is a combinational circuit that performs calculations which can be completed efficiently in one clock cycle. The ALU performs integer additions, subtractions, comparisons, and the like. However, more complex operations, such as multiplication, division, and all floating point operations, require more than one clock cycle, as well as some additional state (registers), and are typically performed in functional units.

```
-- Sigma16 M1 circuit: ALU.hs
-- John T. O'Donnell, 2021
-- See Sigma16/README and https://jtod.github.io/home/Sigma16/

------------------------------------------------------------------------
--Arithmetic/Logic Unit
------------------------------------------------------------------------

module Circuit.ALU where

-- Arithmetic and logic unit for the M1 processor circuit, which implements
-- the core subset of Sigma16.

import HDL.Hydra.Core.Lib
import HDL.Hydra.Circuits.Combinational

{- The ALU calculates a function of word inputs x and y (which are
usually the contents of two registers) and the condition code cc (the
contents of R15).

(The Core architecture doesn't need the cc as an input to the ALU, but
the more advanced instructions do, and cc is provided as an input to
```

simplify future zetension of M1.)

The ALU produces a word output r, which is a
numeric result (typically loaded into the destination register), and a
comparison result ccnew which is the new value to be loaded into the
condition code register. The ALU performs addition, subtraction,
negation, increment, and comparision. The function is determined by
control signals (alua, alub, aluc).

```
Control inputs:
   alua, alub, aluc
Data inputs:
   x
   y
   cc
Data outputs:
   r      = function (a,b,c) x y cc
   ccnew  = compare x y cc
```

The data output r is the result of an arithmetic operation which is
determinted by the control inputs:

op = (alua,alub,aluc)	
a b c	r
0 0 0	x+y
0 0 1	x-y
0 1 0	-x
0 1 1	x+1
1 0 0	cmp

The condition code flags are as follows. Sigma16 indexes bits
starting from 0 in the rightmost (least significant) position, so the
flags cluster toward the right end of a word. (Note: The Hydra !!
operator indexes bits from the left.)

bit index	Relation	Symbol
0	> Int	>
1	> Nat	G

```
|          2 | =               | =      |
|          3 | < Nat           | L      |
|          4 | < Int           | <      |
|          5 | Int overflow    | v      |
|          6 | Nat overflow    | V      |
|          7 | Carry           | C      |
|          8 | Stack overflow  | S      |
|          9 | Stack underflow | s      |
-}
```

5.1.1 Interface to ALU

The ALU calculates a function of word inputs x and y (which are usually the contents of two registers) and cc (the contents of R15). It produces a word output r, which is a numeric result (typically loaded into the destination register), and a comparison result ccnew which is the new value to be loaded into the condition code register. The ALU performs addition, subtraction, negation, increment, and comparision. The function is determined by control signals (alua, alub, aluc).

- Control inputs:
  - `alua`, `alub`, `aluc` are three bits that form an operation code for the ALU
- Data inputs:
  - `x` is the first 16-bit operand word which can represent either a natural number (represented in binary) or an integer (represented in two's complement).
  - `y` is the second 16-bit operand word. It should have the same type (integer or natural) as `x`.
  - `cc` is the current value of the condition code, which is the value in R15. (Note: the current version of M1 does not actually use `cc`, but this input is provided for extension of M1 to handle some of the more advanced instructions which do require it.)
- Data outputs:
  - `r` is the 16-bit result, calculated as a function of `x` and `y`. The particular function is determined by the `(alua, alub, aluc)` operation code (see table below).

  – `ccnew` is a 16-bit word comprising a set of flags indicating comparisons, overflow, and other conditions (see table below). A *flag* is a Boolean represented as a bit, and the condition code is a word containing all the flags required by the Sigma16 architecture.

An integer is a whole number that can be negative, zero, or positive. Integers are sometimes called "signed integers". A natural number is a whole number that can be zero or greater than zero. Natural numbers are sometimes called "counting numbers".

Both types — integers and naturals — are essential in computing. Most programming languges typically provide integer variables. Machine language programs perform calculations on addresses, which are natural numbers (addresses of memory locations start at 0 and count up). It may seem to a programmer that nearly all whole number arithmetic uses signed integers, but the reality is the opposite, computers perform far more arithmetic on natural numbers than on integers.

The M1 ALU performs both integer and natural number arithmetic. The only difference between addition and subtraction of integers and naturals is in the treatment of overflow. A 16-bit word can represent either

- a binary natural nunber x such that $0 \leq x < 2^{16} - 1$
- a two's complement signed integer x such that $-2^{15} \leq x < 2^{15}$

If you add two positive integers that are small enough so their sum isn't too large, an integer addition will get the same result as a natural addition, regardless of whether these numbers had a signed integer type or an unsigned natural type.

However, if you add two positive integers x and y where $x + y \geq 2^{15}$, then the result is correct for natural number addition, but an overflow for two'c complement.

The ALU performs addition for either natural numbers or integers; the result is the same. It calculates comparisons and overflow flags separately for naturals and integers, as these are different. The flags go into `ccnew`, the new value of the condition code.

The data output `r` is the result of an arithmetic operation which is determinted by the control inputs, op = (alua, alub, aluc)

a b c	r
0 0 0	$x + y$
0 0 1	$x - y$
0 1 0	$-x$
0 1 1	$x + 1$
1 0 0	cmp

Sigma16 provides condition code flags indicating the results of comparisons and a number of error conditions. Each flag has a specific bit index in the condition code, given in a table below.

Equality is the same for integers and naturals: two numbers are equal if and only if all their bits are the same. However, there are separate $>$ conditions for integers and naturals, and separate $<$ conditions as well. Furthermore, overflow is different for integers and naturals.

For each condition, the table gives the corresponding bit index, and also a symbol which is used in the Sigma16 emulator to present the condition code in a more readable form.

- For example, suppose a comparison is being made between 5 and 6, using either binary or two's complement: the result will be the same. To do this, the operation code is 100 to indicate cmp, $x = 5$, and $y = 6$. Now $5 < 6$ for both integers and naturals, so `ccnew` must have a 1 at index 3 and index 4. The condition code indices start at 0 from the least significant position. The hex value of the resulting condition code is 0018.

- Suppose an integer comparison is being made between $x = -3$ and $y = 2$. (This has to be interpreted as an integer (two's complement) operation because -1 cannot be represented in binary.) The words representing these operands are hex fffd and 0002. The ALU gives an integer comparison result of $x < y$ (bit index 4). Since the ALU always does both operations — binary and two's complement — it also produces the binary comparison result of $x > y$ (bit index 1). So the final condition code result is 0012, or `<G`

bit index	Relation	Symbol
0	$>$ Int	g
1	$>$ Nat	G
2	$=$	$=$
3	$<$ Nat	L
4	$<$ Int	$<$
5	Int overflow	v
6	Nat overflow	V
7	Carry	C
8	Stack overflow	S
9	Stack underflow	s

5.1.2 Running test data on the ALU

The ALU can be tested and demonstrated on its own, with a simulation driver `ALUrun` that provides its inputs. To run the test data, go to the M1 directory and enter this command:

```
ghc -e main Circuit/ALUrun
```

5.1.3 The ALU circuit

The ALU is a combinational circuit: it contains no flip flops, and it performs every calculation in one clock cycle.

The circuit is essentially a ripple carry binary adder, along with a ripple carry binary comparator. The ripple circuits have a gate delay of $O(n)$ for n-bit words. There are more advanced circuits that produce the same results with a gate delay of only $O(\log n)$.

The ALU must calculate several functions, but it's inefficient to implement each with a separate circuit. Also, the Core subset of Sigma16 requires only a few functions. Commercial computers may have dozens, and it would be wasteful to do them all with separate circuits.

The approach is to make the binary adder perform subtractions as well as additions, by pre-processing‘ the x and y inputs. Similarly, the results of the binary comparator are post-processed to derive integer (two's complement) comparison results.

There are some important points to understand about the circuit.

- This is a circuit specification. It is *not* a computer program.

- The circuit consists entirely of logic gates and wires connecting the logic gates. There are no programming language statements. The circuit is hardware, not software.
- In principle, you could draw a schematic diagram showing all the gates and wires, but this would be unreadable since there are so many components. Using a textual hardware description language enables us to describe the circuit more concisely and readably. Schematic diagrams are fine for toy examples, and they are also ok for vague block diagrams that give a general idea but omit most of the technical content. Hardware description languages are better when you want to keep *all* the technical detail yet still have a readable description.
- The specification consists of equations. There are no assignment statements, or any other "effects".
- As always in mathematics, the order that you write down equations is immaterial. The ALU does not "execute" the equations from the first to the last. The equations are "timeless"; they merely say that values are the same.
- All the logic gates are operating in parallel, all the time. A logic gate does not wait for its inputs to become valid.
- But validity of signals propagates through the circuit according to data dependence and gate delay. For example, consider `inv (xor2 carry msb)`. The output of the inverter becomes valid one gate delay after its input becomes valid, and that happens one gate delay after both of `carry` and `msb` become valid.

Defining equation for ALU. The inputs are the operation code of three bits, the words x and y, and the current value of the condition code cc (which won't actually be used). The outputs are the result and ccnew, the new value of the condition code.

```
alu n (alua,alub,aluc) x y cc = (result, ccnew)
  where
```

Constant words. The circuit needs a word `wzero` with all bits 0, and another word `wone` with the rightmost bit 1 and all other bits 0.

```
-- Constant words
    wzero = fanout n zero
    wone = boolword n one
```

Determine type of function being evaluated. If all the functions calculated by the ALU were unrelated to each other, we would just use a demux to decode the 3-bit operation code. However, the circuit uses just two key calculation circuits (adder and comparator) and it uses preprocessing of the inputs to the adder post processing of the output from the comparator. These operations are closely related. So our approach to decoding the operation code is to define three key signals that determine what is going on:

- `arith` is 1 if the circuit is doing an arithmetic operation, such as addition, subtraction, or incrementing.
- -negating- is 1 if the circuit needs to compute $x - y$ instead of $x + y$.
- `comparing` is 1 if the circuit is doing a comparison but not an arithmetic operation.

```
-- Determine type of function being evaluated
    arith = inv alua  -- doing arithmetic operation, alu abc one of 000 001 010 100
    negating = and2 (inv alua) (xor2 alub aluc)  -- alu abc = 001 or 010
```

Prepare inputs to adder. Although the ALU receives inputs x and y, these are not passed directly to the adder. Instead, they receive some preprocessing in order to make the adder perform the desired operation. This is accomplished by calculating new values `x'` and `y'`, which are the actual inputs to the adder circuit.

- 00. $r = x + y$ so we can pass x and y to the adder by defining $x' = x$‘ and $y' = y$, and setting the carry input to 0.
- 01. $r = x - y$. To make the adder do the subtraction, we need to invert y and set the carry input to 1. So $x' = x$ and $y' = invwy$. (The `invw` circuit takes a word and inverts all its bits.)
- 10. $r = -x$. We compute $-x = 0 - x$, so we set $x' = 0$, $y' = invwx$, and set the carry input to 1.
- 11. $r = x + 1$. Define $x' = x$ and $y' = 1$ and set the carry input to 0. (An alternative approach would be to set $y' = 0$ and set the carry input to 1; it makes little difference.)

```
-- Prepare inputs to adder
    x' = mux2w (alub,aluc) x x wzero x
    y' = mux2w (alub,aluc) y (invw y) (invw x) wone
```

The adder. The word inputs to the adder are x' and y', the results of the preprocessing described above. The carry input is `negating` because a subtraction requires that 1 is added to the sum. We also define `msb` to be the most significant bit of the result; this will give the sign.

```
-- The adder
    xy = bitslice2 x' y'
    (carry,result) = rippleAdd negating xy
    msb = result!!0 --- most significant bit of result
```

Binary comparison. The ripple comparator gives three bits indicating whether $x < y$, $x = y$, or $x > y$. These are binary (natural) comparisons.

```
-- Binary comparison
    (lt,eq,gt) = rippleCmp xy
```

Two's complement comparison. The ALU also needs to work out the comparison relation for integers. These are derived from the comparison for naturals, which is output by the ripple comparator. There are four cases, depending on the leftmost (most significant) bits of the operands x and y. Those two bits are obtained by taking the leftmost pair of bits from xy. In all cases, equality is the same for integers and natural numbers, so we define `eq_tc = eq`.

- 00. Both x and y are nonnegative, so $x < y$ as integers if and only if $x < y$ as natural numbers, and similar for $x > y$.
- 01. x is nonnegative but y is negative, so $x > y$ as integers.
- 10. x is negative but y is nonnegative, so $x < y$ as integers.
- 11. Both x and y are negative, so $x < y$ as integers if and only if $x < y$ as natural numbers, and similar for $x > y$.

```
-- Two's complement comparison
    lt_tc = mux2 (xy!!0) lt zero one lt
    eq_tc = eq
    gt_tc = mux2 (xy!!0) gt one zero gt
```

```
lt_tc = mux2 (xy!!0) lt zero one lt
eq_tc = eq
gt_tc = mux2 (xy!!0) gt one zero gt
```

Carry and overflow. For natural numbers, there is an overflow if the carry output is 1. For integers, there is an overflow if the most significant bit differs from the carry output.

```
-- Carry and overflow
    mx = x' !! 15      -- sign bit of first operand
    my = y' !! 15      -- sign bit of second operand
    mr = result !! 15  -- sign bit of result
    intovfl = or2 (and3 mx my (inv mr))
                  (and3 (inv mx) (inv my) mr)
    natovfl = carry
    noOvfl  = inv intovfl
```

Relation of integer result to 0. The Sigma16 architecture specifies that arithmetic operations set the condition code to indicate the relation between the result and 0. Thus if the result of calculating $x + y$ is negative, then the $<$ condition is set.

```
-- Relation of integer result to 0
    any1 = orw result                  -- 1 if any bit in result is 1
    neg  = and3 noOvfl any1 msb        -- ok, result < 0
    z    = and2 noOvfl (inv any1)      -- ok, result is 0
    pos  = and3 noOvfl any1 (inv msb)  -- ok, result > 0
```

```
any1 = orw result            -- 1 if any bit in result is 1
neg  = and3 noOvfl any1 msb        -- ok, result < 0
z    = and2 noOvfl (inv any1)      -- ok, result is 0
pos  = and3 noOvfl any1 (inv msb)  -- ok, result > 0
```

Overflow flags: don't indicate overflow for a comparison operation. The overflow conditions should not be set in the condition code if a comparison operation is being performed.

```
-- Overflow flags:  don't indicate overflow for a comparison operation
    fcarry   = and2 arith carry
    fnatovfl = and2 arith natovfl
    fintovfl = and2 arith intovfl
```

```
fcarry   = and2 arith carry
fnatovfl = and2 arith natovfl
fintovfl = and2 arith intovfl
```

Comparison flags: for arithmetic, indicate comparison with 0. The flags for the conditions are defined, depending on whether the ALU is performing an arithmetic operation or a comparison.

```
-- Comparison flags: for arithmetic, indicate comparison with 0
    flt      = mux1 arith lt    zero
    flt_tc   = mux1 arith lt_tc neg
    feq      = mux1 arith eq    z
    fgt      = mux1 arith gt    pos
    fgt_tc   = mux1 arith gt_tc pos
```

```
flt      = mux1 arith lt    zero
flt_tc   = mux1 arith lt_tc neg
feq      = mux1 arith eq    z
fgt      = mux1 arith gt    pos
fgt_tc   = mux1 arith gt_tc pos
```

Generate the condition code. The `ccnew` result is a word consisting of the condition flags, with the other bits set to zero.

```
-- Generate the condition code
    ccnew = [ zero,   zero,     zero,     zero,    -- bit 15 14 13 12
              zero,   zero,     zero,     zero,    -- bit 11 10  9  8
              fcarry, fnatovfl, fintovfl, flt_tc,  -- bit  7  6  5  4
              flt,    feq,      fgt,      fgt_tc   -- bit  3  2  1  0
            ]
```

```
ccnew = [ zero,   zero,     zero,     zero,    -- bit 15 14 13 12
          zero,   zero,     zero,     zero,    -- bit 11 10  9  8
          fcarry, fnatovfl, fintovfl, flt_tc,  -- bit  7  6  5  4
          flt,    feq,      fgt,      fgt_tc   -- bit  3  2  1  0
        ]
```

5.1.4 Simulation driver: ALUrun

```
-- Sigma16: ALUrun.hs
-- John T. O'Donnell, 2021
-- See Sigma16/README and https://jtod.github.io/home/Sigma16/

---------------------------------------------------------------------------
```

```
-- Simulation driver and test data for ALU
---------------------------------------------------------------------------

-- Usage: cd to the M1 directory and enter ghc -e main Circuit/ALUrun

module Main where
import HDL.Hydra.Core.Lib
import Circuit.ALU

{-

Result function

| a b c |    r     |
|-------+-----------
| 0 0 0 |   x+y    |
| 0 0 1 |   x-y    |
| 0 1 0 |    -x    |
| 0 1 1 |   x+1    |
| 1 0 0 |   cmp    |

Condition code

| bit index | Relation        | Symbol |
|-----------+-----------------+--------|
|         0 | > Int           | g      |
|         1 | > Nat           | G      |
|         2 | =               | =      |
|         3 | < Nat           | L      |
|         4 | < Int           | <      |
|         5 | Int overflow    | v      |
|         6 | Nat overflow    | V      |
|         7 | Carry           | C      |
|         8 | Stack overflow  | S      |
|         9 | Stack underflow | s      |
-}

alu_input1 =
--   a  b  c     x      y   cc     Operation  Result
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
  [ "0  0  0     14     15   0"    --   x+y       29
  , "0  0  0    125    590   0"    --   x+y      715
  , "0  0  0     49     15   0"    --   x+y       64
  , "0  0  0     21    -19   0"    --   x+y        2
  , "0  0  0     21    -35   0"    --   x+y      -14
  , "0  0  0   -350     75   0"    --   x+y     -275
  , "0  0  0   -420    -90   0"    --   x+y     -510

  , "0  0  1     49     15   0"    --   x-y       34
  , "0  0  1     15     49   0"    --   x-y      -34

  , "0  1  0     39      0   0"    --   -x       -39
  , "0  1  0     25     70   0"    --   -x       -25

  , "0  1  1     17      0   0"    --   x+1       18
  , "0  1  1    193     52   0"    --   x+1      194

  , "1  0  0      5      5   0"    --  cmp     cc = 0004  =
  , "1  0  0      5      7   0"    --  cmp     cc = 0018  <L
  , "1  0  0      7      5   0"    --  cmp     cc = 0003  >G
  , "1  0  0      5     -1   0"    --  cmp     cc = 0009  >L
  , "1  0  0     -1      5   0"    --  cmp     cc = 0012  <G

   ]


---------------------------------------------------------------------------
-- Simulation driver for ALU
---------------------------------------------------------------------------

main = driver  $ do

-- Word size
  let n =  16

-- Input data
  useData alu_input1

  in_a <- inPortBit "a"
  in_b <- inPortBit "b"
  in_c <- inPortBit "c"
  in_x <- inPortWord "x" n
  in_y <- inPortWord "y" n
  in_cc <- inPortWord "cc" n

  let a = inbsig in_a
  let b = inbsig in_b
  let c = inbsig in_c
  let x = inwsig in_x
  let y = inwsig in_y
  let cc = inwsig in_cc

-- Circuit
  let (r,ccnew) = alu n (a,b,c) x y cc

  format
      [string "Inputs:  ",
       string " abc = ", bit a, bit b, bit c,
       string "\n         x = ", bits x, string " $", binhex x,
       string " (bin:", bindec 5 x, string ")",
       string " (tc: ", bitstc 6 x, string ")",
       string "\n         y = ", bits y, string " $", binhex y,
       string " (bin:", bindec 5 y, string ")",
       string " (tc: ", bitstc 6 y, string ")",
       string "\n       Outputs:  ",
       string "\n         r = ", bits r, tcdec 5 r, string "  $", binhex r,
       string "\n         ccnew = ", bits ccnew, string " $", binhex ccnew,
       string "\n"]

  runSimulation
```

5.2 Register file

A register file is an array of registers that are accessed by a register address. Sigma16 has a register file that contains 16 registers, so the register address is a 4 bit word. Each register holds a 16 bit word.

The register file contains the registers that are directly visible to the assembly language programmer. For example, the instruction `add R1,R2,R3` refers to three elements of the register file. There are additional registers needed by the the system which are not directly visible to programs, including the pc, ir, and adr registers. Those are not part of the register file.

The term "register file" is confusing, because it has nothing to do with files in the file system. A better term might be "register array", but "register file" is the traditional name.

Notation. The assembly language notation uses names like R3, which means the register with index 3 in the register file. Rd means the register whose index is the current value of the 4-bit word `d`, and similar for Rsa and Rsb.

```
module Circuit.RegFile where

import HDL.Hydra.Core.Lib
import HDL.Hydra.Circuits.Combinational
import HDL.Hydra.Circuits.Register

-- Register files
```

5.2.1 Basic register file

The basic register file is a circuit that contains 16 registers, and each register holds a 16 bit word. It has the ability to read out two registers simultaneously, and to prepare to load one of the registers at the next clock tick. This makes it possible to read two operands, run them through the ALU, and prepare to load the result, all during one clock cycle. That enables M1 to execute the SIgma16 `add` instruction in just one cycle, after the instruction has been fetched and decoded. The registers are named R0 to R15, and all the registers work the same way.

To achive all this, the register file needs control inputs that tell it which registers to read out, whether to load a register, and if so which register to load and what value to put into it.

We will start with `regfile1`, which contains registers of 1 bit. The full register file will consist of 16 copies of `regfile1`. The interface provides all of required inputs, and the circuit outputs two registers.

In general, the size of the register file is determined by the word size k of the register addresses. There are three register addresses, `sa`, `sb`, and `d`, so each is a k bit word. The number of registers is 2^k. The register file is generated by a recursion over k. For Sigma16, $k = 4$.

The regfile1 circuit has 2^k words, each consisting of 1 bit. A full register file with n bit words consists of n copies of the regfile1 circuit.

```
regfile1
  :: CBit a  -- a is signal type for synchronous circuit
  => a       -- ld control
  -> [a]     -- d:ds = destination address
  -> [a]     -- sa:sas = source address a
  -> [a]     -- sb:sbs = source address b
  -> a       -- x
  -> (a,a)   -- readout (a = reg[sa], b = reg[sb])
```

The circuit is generated by a recursive definition.

Base case. The base case is $k = 0$, so the register addresses are all empty words `[]`. There is only $2^0 = 1$ register. That register is r. Since there are two readouts, the output is (r, r). That may look strange, but this will happen if both source addresses are the same, which occurs in an instruction like `add R1,R2,R2`.

```
regfile1 ld [] [] [] x = (r,r)
  where r = reg1 ld x
```

Recursion case. The recursion case occurs when $k > 0$, so each of the register addresses has at least one bit. For example, the destination address must be in the form `(d:ds)`, where the initial bit is `d` and the rest of the address is `ds`, whose word size is $k - 1$. The other addresses have similar forms `(sa:sas)` and `(sb:sbs)`. The circuit is generated by defining two half-size register files, each of size $k - 1$. The most significant bit of each address selects one of those two subsystems, and the remainder of the address bits are sent to both subsystems.

```
regfile1 ld (d:ds) (sa:sas) (sb:sbs) x = (a,b)
  where
    (a0,b0) = regfile1 ld0 ds sas sbs x
    (a1,b1) = regfile1 ld1 ds sas sbs x
    (ld0,ld1) = demux1 d ld
    a = mux1 sa a0 a1
    b = mux1 sb b0 b1
```

The full `regfile` circuit is an array of \$n\$-bit words, and it consists simply of n copies of a `regfile1` circuit. In general, there are two size parameters for the generator: the register word size n and the address size k. For M1, $n = 16$ and $k = 4$.

```
regfile :: CBit a => Int -> Int
  -> a -> [a] -> [a] -> [a] -> [a] -> ([a],[a])

regfile n k ld d sa sb x =
   unbitslice2 [regfile1 ld d sa sb (x!!i)  | i <- [0..n-1]]
```

5.2.2 Handling R0 and R15

The basic register file treats all registers the same. However, Sigma16 treats R0 and R15 as special cases:

- R0 is always 0. It is legal to load another value into it, but any readout of R0 will yield 0.
- R15 is the condition code. It holds the result of the `cmp` instruction. Furthermore, arithmetic instructions, which place their result in the register specified by `ir_d`, also set some flags in the condition code indicating overflow and other conditions. Thus some instructions load separate values into two registers. The conditional jump instructions automatically fetch a bit from R15 in order to decide whether to jump.

It would be straightforward but inefficient to implement these requirements by using several clock cycles. For example, an `add` instruction could use one cycle to load the result into the destination register, and a second cycle to load the condition code into R15. But this would severely slow down the machine. Although M1 is not pipelined, we would like to be able to use pipelining in a more complex machine to achieve and effective rate of one cycle per instruction for the RRR instructions. Using a separate cycle to load the condition code would result in a 100% slowdown. Therfore we will design a more sophisticated register file, `regfileSpec`, that can handle the special cases R0 and R15 without slowdown.

There are several ways the circuit could handle R0:

- Treat R0 just like all the other registers in the register file circuit, and use additional logic in the datapath to replace its value by 0 whenever R0 is fetched. This way, R0 could contain any arbitrary value but fetching it would always give 0.

- Use additional logic in the datapath to replace any value loaded into R0 by 0. This way, R0 would always contain 0.
- Design a new register file circuit that doesn't use flip flops to hold the contents of R0, but instead arrange the readout logic to give 0 when R0 is fetched. This way, R0 is a virtual register: it doesn't actually exist but the register file behaves as if R0 contain s0.

The last of those approaches is best, and M1 uses it. Thus R0 does not actually exist: it doesn't contain any flip flops. A load into R0 is discarded, and the combinational logic for fetching a register produces the value 0 when R0 is fetched. This is relatively simple to implement, and the resulting circuit consumes less power and requires less chip area than alternatives that use flip flops to hold the irrelevant state of R0.

R15, which holds the condition code, is more complex, for several reasons:

- The machine may need to read out R15 as well as two source registers at the same time, so there need to be three data word outputs from the circuit.
- The machine may need to load a new value into both R15 and some other destination register at the same time. Therefore there need to be two data word inputs, one for the destination register and one for R15.
- Another complication is that the machine language program could specify R15 as the destination in an instruction that also sets the condition code. For example, `add` sets the condition code to indicate whether overflow occurred. The circuit needs to detect and resolve this conflict, since it cannot load both a data value and a condition code into the same register at the same time.

The Sigma16 archiecture specifies that if the destination register is R15, then the result is loaded into R15 and the condition code is discarded.

- `add R14,R1,R2` places the sum in R14 and the condition code in R15
- `add R15,R1,R2` places the sum in R15 and the condition code is discarded.

It would be possible to handle the condition code by using two clock cycles for arithmetic instructionss, one to put the result into the destination register and another to put the condition code into R15. However, add instructions are frequently executed, and this would give an unacceptable slowdown. The M1 circuit is designed to be as simple as possible, but more advanced circuits should be possible with the architecture.

5.2.3 Interface to register file with special R0 and R15

`regFileSpec` is a register file circuit that is similar to `regFile` except that it provides for the special cases of R0 and R15.

- R0 always outputs 0
- R15 is always output, and can be loaded independently from other registers

The control inputs to the register file need to tell it what to do. There are four possibilities, so two control signals are required to specify one of these four cases: `ld` and `ldcc`.

`ld` = **0,** `ldcc` = **0** *Don't load anything into any register in the register file.* This happens during instruction fetch, during the initial cycles of RX instructions like `load`, and in many other cases.

`ld` = **0,** `ldcc` = **1** *Load a value into the condition code, R15.* For example, `cmp R2,R3` puts the result of the comparison into R15, which is not one of the operands of the instruction. Although the assembly language notation for the `cmp` instruction specifies only the two registers to compare, the instruction is RRR format, and the destination register is ignored. The assembler uses 0 for the destination register, but that is an arbitrary choice and the machine ignores it.

`ld` = **1,** `ldcc` = **0** *Load a value into the destination register, leave R15 unchanged.* For example, `load R2,x[R0]` must load a word into R2 but R15 remains unchanged.

`ld` = **1,** `ldcc` = **1** *Load a value into both the destination register and R15.* For example, `add R3,R4,R5` loads the sum into R3 and the condition code into R15. The condition code for `add` indicates overflow, carry, and the sign of the result.

The circuit produces three output words, `a`, `b`, and `cc`:

- `a` is a readout giving the value of `Rsa`, the register indexed by the `sa` input. The `a` output is produced by a multiplexer tree that uses the `sa` input to select which register to read.

- `b` is a readout giving the value of `Rsb`. It is similar to `a` but using a separate multiplexer tree, so `a` and `b` can be read out independently of each other.
- `cc` is a readout giving the value of R15. There is a direct connection from R15 to the `cc` output; this readout is separate from the multiplexer trees used for `a` and `b`.

The actions to be performed are controlled by `ld`, `ldcc`, and `d`, and the input data values are `x` and `ccnew`:

- if `ld` then `reg[d]` := `x` (In case $d = 0$, there is no effect.)
- if $ldcc \wedge \neg(ld \wedge d = 15)$ then `reg[15]` := `cc`

If both of these conditions are true, so both `Rd` and `R15` are loaded, then $d \neq 15$.

5.2.4 The interface

Inputs

ld 1 bit, load control. Determines whether the destination register will perform a load: if `ld=1` then `reg[d]` : x=

ldcc 1 bit, R15 load control. Determines whether xcc will be loaded into R15

x 16 bits, data input to be loaded into destination register

xcc 16 bits, new condition code. Data input to be loaded into R15

d 4 bits, destination address

sa 4 bits, source a address

sb 4 bits, source b address

Outputs

a 16 bits. Contents of Rsa

b 16 bits. Contents of Rsb

cc 16 bits. Contents of R15

```
{- Register file with special treatment of R0 and R15

reg[0] always outputs 0
reg[15] is always output, and can be loaded independently from other registers

Inputs
  ld     load control
  ldcc   load into R15
  x      data input
  xcc    R15 data input
  d      destination address
  sa     source a address
  sb     source b address

Outputs
  a = reg[sa]
  b = reg[sb]
  cc = reg[15]

Effect on state (from programmer's perspective)
  if ld                  then reg[d] := x
  if ~(ld & d=15) & ldcc  then reg[15] := xcc

State update (from perspective of circuit)
  reg[0] there is no state
  reg[d] for 0 < d < 15:  reg[d] := if ld then x else reg[d]
  reg[15] := if ld & d=15 then x
               else if ldcc then xcc
               else reg[15]
-}
```

5.2.5 Implementing the register file with special cases

The `regfileSpec1` circuit is mostly the same as the basic `regfile`, except for the way it handles the special cases R0 and R15.

To design the circuit, we need to work out the new value that each register will receive at the next clock tick:

- R0 has no state
- For $0 < d < 15$, Rd := if `ld` then `x` else `Rd`

- `R15` := if `ld` $\wedge$ `d` = 15 then `x` else if `ldcc` then `xcc` else `R15`

The if-then-else notation used to describe the new state will be implemented using simple combinational logic. There is no sequence of commands. These are if-then-else *expressions*, not statements.

The circuit is defined by a recursion over the addresses. Each of the register addresses is a 4-bit word. (If the lengths of the addresses are not the same there will be a pattern match error.)

The circuit is generated recursively. The recursion has the same structure as the generator fot the basic rebister file: it is controlled by the register address operands, which must have the same word size.

The key issue is dealing with R0 and R15. The recursive definition means that the generator will be called for a number of sequences of consecututive registers. The full circuit will be called to generate R0,...,R15. This will make two recursive calls, one for R0,...,R7 and another for R8,...,R15.

As the recursion unfolds, each sequences of registers to be generated falls into one of these categories:

- *RFfull.* The full register file is being generated, and it includes both R0 and R15.
- *RFhead.* The sequence of registers being generated is the "head" of the sequence because it includes R0 (but not R15). This happens when generating R0,...,R7 and R0,...,R3 and R0,R1 and finally just R0.
- *RFtail.* The sequence of registers being generated is the "tail" of the sequence because it includes R15 (but not R0). This happens when generating R8,...,R15 and R12,...,R15 and R14,R15 and finally just R15.
- *RFinside.* The sequence of registers is "inside" the full register file, and it doesn't include either R0 or R15. This happens, for example, when generating R4,...,R7 and in many other cases.

If you draw a tree diagram showing how the recursion unfolds, the sequences reached by always taking the left branch will all be `RFhead`, and the sequences reached by always taking the right branch will all be `RFtail`. All the other sequences are `RFinside`.

We define symbols for the four cases:

```
-- RFspan determines how to generate the circuit for the base cases
```

```
data RFspan
  = RFfull      -- contains R0
  | RFhead       -- contains R15
  | RFtail       -- contains R15
  | RFinside     -- contains neither R0 nor R15
```

As the recursive generator expands the circuit, each invocation of `regFileSpec1` is given its own span (`RFhead` etc.), and it needs to specify the span of the two sub-circuits it creates.

```
headType, tailType :: RFspan -> RFspan
headType RFfull   = RFhead
headType RFinside = RFinside
headType RFhead   = RFhead
headType RFtail   = RFinside

tailType RFfull   = RFtail
tailType RFinside = RFinside
tailType RFhead   = RFinside
tailType RFtail   = RFtail
```

The type of the circuit generator takes the span as the first argument. The remaining arguments are the inputs to the circuit.

```
regFileSpec1
  :: CBit a
  => RFspan
  -> a           -- ld: if ld then reg[d] := x
  -> a           -- ldcc: if ldcc then reg[15] := xcc (but ld R15 takes precedence)
  -> [a]         -- d: destination address
  -> [a]         -- sa: source a address
  -> [a]         -- sb: source b address
  -> a           -- x = data input for reg[d]
  -> a           -- xcc = data input for condition code R15
  -> (a,a,a)     -- (reg[sa], reg[sb], reg[15])
```

The generator is defined by recursion over the address words; its structure is the same as the generator for the basic `regfile` circuit. There will be a pattern match error if the addresses (d, sa, sb) don't all have the same number of bits.

Base cases. The whole point of the `RFspan` argument is to tell the generator how to make a singleton register, when the base case of the recursion is reached.

- *RFinside.* This is an ordinary register: it cannot be either R0 or R15. It is defined exactly the same way as all the registers in the basic `regfile` circuit.
- *RFhead.* This register is R0. All the outputs are set to `zero`: this is the place where the readout of R0 is guaranteed to be 0.
- *RFtail.* This is R15. A register `r` is generated.
  - The value of `r` is placed on all the output ports.
  - The load control of `r` is defined to be 1 if either `ld` or =ldcc- is 1.
  - The data input to `r` is set to either `x` (the value to be loaded into the destination register) or `xcc` (the new value of the condition code).
- *RFfull.* This case cannot occur, as a singleton register cannot be both R0 and R15. It would generate an error message.

```
regFileSpec1 RFinside ld ldcc [] [] [] x xcc = (r,r,zero)
  where r = reg1 ld x
regFileSpec1 RFhead ld ldcc [] [] [] x xcc = (zero,zero,zero)
regFileSpec1 RFtail ld ldcc [] [] [] x xcc = (r,r,r)
  where r = reg1 (or2 ld ldcc) (mux1 ld xcc x)
regFileSpec1 RFfull ld ldcc [] [] [] x xcc = error "impossible case"
```

Recursion case. Two smaller register files are generated, one for the case where `d` = `sa` = `sb` = 0, and another for `d` = `sa` = `sb` = 1. This is almost identical to the recursion case for the basic `regfile`. The only difference is in handling the span type argument `rft`.

- The span for the low-address sub-circuit is `headType rft`. This ensures that the smaller circuit will preserve the head property if this entire circuit is a head; otherwise it will be an inside.
- The span for the high-address subcircuit is `tailType rft`. This ensures that the smaller circuit will be either a tail or an inside.

```
regFileSpec1 rft ld ldcc (d:ds) (sa:sas) (sb:sbs) x xcc = (a,b,cc)
  where (a0,b0,cc0) = regFileSpec1 (headType rft) ld0 ldcc ds sas sbs x xcc
        (a1,b1,cc1) = regFileSpec1 (tailType rft) ld1 ldcc ds sas sbs x xcc
        (ld0,ld1) = demux1 d ld
        a = mux1 sa a0 a1
        b = mux1 sb b0 b1
        cc = cc1
```

The full register file with \$n\$-bit words is built from n copies of the `regFileSpec1` circuit.

```
-- n-bit register file with special cases for R0 and R15

regFileSpec
  :: CBit a
  => Int              -- word size
  -> a                -- ld: if ld then reg[d] := x
  -> a                -- ldcc: load R15
  -> [a]              -- d: destination address
  -> [a]              -- sa: source a address
  -> [a]              -- sb: source b address
  -> [a]              -- x = data input for reg[d]
  -> [a]              -- xcc = data input for condition code R15
  -> ([a],[a],[a])  -- (reg[sa], reg[sb], reg[15])

regFileSpec n ld ldcc d sa sb x xcc =
  unbitslice3 [regFileSpec1 RFfull ld ldcc d sa sb (x!!i) (xcc!!i)
                 | i <- [0 .. n - 1]]
```

5.3 Main datapath

The main data path provides the computational capabilitties required. It includes the registers, the ALU, and signals that connect these components.

M1 currently leaves the multiply and divide instructions unimplemented, but functional units for those operations would also be added to the data path.

The datapath does not determine *when* the various operations will actually occur. That is determined by a set of control signals which are inputs to the datapath.

```
-- Sigma16 M1: Datapath.hs
-- John T. O'Donnell, 2021
-- See Sigma16/README and https://jtod.github.io/home/Sigma16/

{-# LANGUAGE NamedFieldPuns #-}
{-# LANGUAGE RecordWildCards #-}

module Circuit.Datapath where

-- This module defines the datapath for the M1 circuit, a processor
-- for the Sigma16 architecture.

-- The datapath contains the registers, computational systems, and
-- interconnections.  It has two inputs: a set of control signals
-- provided by the control unit, and a data word from the either the
-- memory system or the DMA input controller.

import HDL.Hydra.Core.Lib
import HDL.Hydra.Circuits.Combinational
import HDL.Hydra.Circuits.Register

import Circuit.Interface
import Circuit.ALU
import Circuit.RegFile

datapath
  :: CBit a
  => CtlSig a
  -> SysIO a
  -> [a]
  -> DPoutputs a

datapath (CtlSig {..}) (SysIO {..}) memdat = dp
  where

-- Interface
    dp = DPoutputs {..}

-- Size parameters
    n = 16    -- word size
```

The datapath contains a register file and instruction control registers pc, ir, and ad.

In designing a complex system it's a good idea to take a "divide and conquer" approach, where you break the big design down into separate smaller ones. These register definitions use that approach.

For example, what should the data input to register file be? That's a complicated question, and it involves looking at the whole instruction set architecture. For now, just invent a name p for the data input. That way we can defer figuring out how to generate p until later, and for now just get on with defining the registers. Similarly, names q and u are invented for the inputs to the pc and ad registers respectively.

On the other hand,the data input to ir can be defined to be memdat, because the only time the processor will put anthing into the instruction register is when it is fetching an instruction word from the memory.

```
-- Registers
    (a,b,cc) = regFileSpec n            -- size parameter
                 ctl_rf_ld ctl_rf_ldcc  -- load controls
                 ir_d rf_sa rf_sb       -- register addresses
                 p ccnew                -- data inputs
    ir = reg n ctl_ir_ld memdat
    pc = reg n ctl_pc_ld q
    ad = reg n ctl_ad_ld u
```

We invent the names x and y for the data inputs to the ALU. Sometimes those will be set to the a and b readouts from the register file (as in the RTM circuit), but sometimes we will need to provide different inputs to the ALU. For now we just call the inputs x and y, and the details can be defined later.

```
-- ALU
    aluOutputs = alu n (ctl_alu_a, ctl_alu_b, ctl_alu_c) x y cc
    (r,ccnew) = aluOutputs
```

Now we can work through what the various signal values need to be. This is done by considering the entire instruction set architecture and working out all the cases that can occur. What typically happens is that some input to a subsystem should be a choice between several alternatives, and to make the choice we introduce a multiplexer.

For example, the add and sub instructions require the first input to the ALU to be Ra, the register selected by the sa field of the ir. But sometimes we are using the ALU to increment the pc, and that requires x to be pc. So the definition of `x` uses a `mux1w` to choose between `a` and `pc`. The control input to the multiplexer is `ctl_x_pc`, and that signal will be defined by the control circuit. The datapath just provides the *ability* to connect either `a` or `pc` into the `s` input to the ALU, but the control will decide which choice to make by defining `ctl_x_pc`.

The rest of the internal signals are defined in a similar way, by working through the operations that need to be supported by the datapath.

There is a pattern in these definitions: each signal must be chosen from a set of alternatives, depending on what the processor is doing. The choice is implemented by a multiplexer and the control circuit will determine which choice is made.

```
-- Internal processor signals
    x = mux1w ctl_x_pc a pc             -- alu input 1
    y = mux1w ctl_y_ad b ad             -- alu input 2
    rf_sa = mux1w ctl_rf_sd ir_sa ir_d  -- a = reg[rf_sa]
    rf_sb = mux1w (and2 io_DMA io_regFetch)
              ir_sb
              (field io_address 12 4)
    p  = mux1w ctl_rf_pc                -- regfile data input
           (mux1w ctl_rf_alu memdat r)
           pc
    q = mux1w ctl_pc_ad r ad        -- input to pc
    u = mux1w ctl_ad_alu memdat r   -- input to ad
    ma = mux1w ctl_ma_pc ad pc      -- memory address
    md = a                          -- memory data
```

The datapath also defines names for the fields of an instruction. The following definitions don't contain any logic gates at all; they are just wiring patterns.

```
-- Instruction fields
    ir_op = field ir  0 4           -- instruction opcode
    ir_d  = field ir  4 4           -- instruction destination register
    ir_sa = field ir  8 4           -- instruction source a register
    ir_sb = field ir 12 4           -- instruction source b register
```

5.4 Multiplier functional unit

```
-- Multiply: circuit that multiplies two binary natural numbers
-- This file is part of Hydra, see Hydra/README.md for copyright and license
```

```
----------------------------------------------------------------------
-- Binary multiplier circuit
----------------------------------------------------------------------

module Circuit.Multiply where
import HDL.Hydra.Core.Lib
import HDL.Hydra.Circuits.Combinational
import HDL.Hydra.Circuits.Register

-- Definition of a circuit that multiples two binary integers.  The
-- circuit is a functional unit, which uses a start control signal to
-- initiate a multiplication and produces a ready output signal to
-- indicate completion.

-- The multiplier circuit uses the sequential shift-and-add algorithm
-- to multiply two k-bit binary numbers, producing a 2k-bit product.
-- The specification is general, taking a size parameter k::Int.  The
-- start control signal tells the multiplier to begin computing the
-- product x*y, and any other multiplication in progress (if any) is
-- aborted.  In order to make the simulation output more interesting,
-- the multiplier outputs its internal register and sum values as well
-- as the ready signal and the product.

multiply
  :: CBit a                -- synchronous circuit
  => Int                   -- k = input word size; output is 2*k
  -> a                     -- start = control input
  -> [a]                   -- x = k-bit word
  -> [a]                   -- y = k-bit word
  -> (a,[a],[a],[a],[a])   -- (ready, product, ...internalsignals...)

multiply k start x y = (ready,prod,rx,ry,s)
  where
    rx = latch k (mux1w start (shr rx) x)
    ry = latch (2*k)
             (mux1w start
                (shl ry)
                (fanout k zero ++ y))
    prod = latch (2*k)
```

```
             (mux1w start
                (mux1w (lsb rx) prod s)
                (fanout (2*k) zero))
    (c,s) = rippleAdd zero (bitslice2 ry prod)
    ready = or2 (inv (orw rx)) (inv (orw ry))
```

5.4.1 Simulation driver: MultiplyRun

```
-- MultiplyRun: simulation driver for multiply circuit
-- John O'Donnell, 2021
-- See Sigma16/README and https://jtod.github.io/home/Sigma16/

-- To run the multiplier:
--   $ ghci
--   ghci> :load MultiplyRun
--   ghci> :main
--   hydra> run

module Main where
import HDL.Hydra.Core.Lib
import Circuit.Multiply

mult_test_data_1 :: [String]
mult_test_data_1 =
--     start  x    y
--     ~~~~~~~~~~~~~~
       ["1    50   75",  -- start=1 to multiply 50 * 75, expect 3750
        "0    0     0",  -- start=0 to give circuit time
        "0    0     0",  -- a number of clock cycles are needed
        "0    0     0",  -- give it another cycle
        "0    0     0",
        "0    0     0",
        "0    0     0",
        "0    0     0",
        "0    0     0",
        "1  100   100",  -- start=1 to multiply 100*100, expect 10000
        "0    0     0",
        "0    0     0",
        "0    0     0",
```

```
        "0    0     0",
        "0    0     0",
        "0    0     0",
        "0    0     0",
        "0    0     0",
        "0    0     0",
        "0    0     0",
        "1  100   100",  -- start=1 to multiply 100*100, expect 10000
        "0    0     0",  -- working on 100*100
        "0    0     0",  -- working on 100*100
        "1    2     3",  -- abort and start multiplying 2*3
        "0    0     0",
        "0    0     0",
        "0    0     0",
        "0    0     0",
        "0    0     0",
        "0    0     0"]

-- main :: Driver a
main :: IO ()
main = driver $ do

-- Input data
  useData mult_test_data_1

-- Size parameter
  let k = 8

-- Input ports
  in_start <- inPortBit "start"
  in_x     <- inPortWord "x" k
  in_y     <- inPortWord "y" k

-- Input signals
  let start = inbsig in_start
  let x     = inwsig in_x
  let y     = inwsig in_y

-- Circuit
  let (rdy,prod,rx,ry,s) = multiply k start x y
```

```
-- Format the signals
  format
    [string "Input: ",
     bit start, bindec 4 x, bindec 4 y,
     string "  Output: ",
     bit rdy, bindec 6 prod, bindec 4 rx, bindec 6 ry,
     bindec 6 s,
     string "\n"]

  runSimulation
```

6 Control

6.1 Basic delay elemeent method for control

There are many ways to synthesize a control circuit from a control algorithm. A simple approach is the delay element method.

For example, consider the chain of states for the load instruction. In the basic delay element method (which doesn't provide for DMA cycle stealing), the states would be defined like this, and The control signals are generated directly from those states:

```
dff_load0 = dff (pRX!!1)
dff_load1 = dff st_load0
dff_load2 = dff st_load1
```

The control signals are generated by the states. For example, suppose

- State `dff_load0` asserts `c1` and `c2`
- State `load1` asserts `c3`
- State `load2` asserts `1c` and `c3`

Then the controls are defined by

```
c1 = orw [dff=load0, dff_load2]
c2 = orw [dff_load1]
c3 = orw [dff_load0, dff_load2]
```

6.2 Enhanced delay elements for DMA and cycle stealing

Direct memory access (DMA) is a method for supporting Input/Output. An input operation requires data from an input device to be stored into the memory. An output operation is the reverse: data must be fetched from memory and sent to an output device.

One way to implement I/O is to require the CPU to perform the memory accesses during an I/O operation. This method was actually used on some very early computers (1940s), but it is extremely slow, and is not used on modern computers.

DMA is far more efficient. The idea is that the processor doesn't access the memory for I/O. Instead, it makes a request for input or output; this means simply sending a small message to the I/O system. The I/O system then performs its own accesses to the memory.

For example "print 80 characters in memory starting at address 2bc3". Sigma16 makes this request using a trap instruction: trap R1,R2,R3 specifies the operation by a number in R1, and arguments in R2 and R3. For example, if R1 contains 1 (the trap code for write), thhis tells the I/O system to print the contents of memory starting at the address in R2, and the number of characters is given in R3.

The main technical issue in DMA is that both the processor and the I/O system are making accesses to the memory, and these are likely to happen at the same time. The memory itself, however, can do only one operation at a time. Therefore it is necessary to ensure that the processor and I/O do not interfere with each other.

Suppose the I/O controller needs to fetch a memory location x. To do so, the system needs to set some control signals, and place x on the memory address control. But these actions could interfere with normal execution of the processor. If the processor happens to be accessing memory at some other address, there will be a conflict.

How can we resolve a confliict between the I/O system and the processor when both want to access memory at the same time? There are two general approaches: cycle stealing and a separate memory management unit. The M1 system uses cycle stealing.

The idea behind cycle stealing is that during every clock cycle, either the processor or the memory is performing an action, but never both. In this context, "action" means changing the state by putting new values into the flip flops.

The main system controller provides a signal DMA that indicates whether the processor or the I/O can perform a memory operation during the current cycle. If DMA is 0 the processor should operate normally. If DMA is 1 the I/O system can access the memory, and the CPU should leave its state unchanged at the next clock tick.

In the basic delay element method, the system will definitly set all the control signals corresponding to the current state. However, we need to

- Set all the processor's control signals to 0 during a cycle when DMA=1.
- Leave the control state unchanged at the next clock tick. That enables the processor to retry its current operation in the next clock cycle.

To achieve this, two signals are defined for every state: a flip flop which represents "the processor is trying to be in this statee, unless the cycle has been stolen", and a logic signal that means "the processor is in charge this cycle and is actually generating control signals".

This To support cycle stealing, there is a dff for each state (e.g. `dff_load0`) and an additional signal (`st_load0`) that is 1 if the flip flip is 1 and the cycle is not stolen. If `dff_load0` is 1 it means that the processor needs to execute this state. If `st_load0` is 1 it means the processor actually is in this state and its control signals can be asserted.

```
dff_load0 = dff (or2 (pRX!!1) (and2 dff_load0 io_DMA))
st_load0  = and2 dff_load0 cpu
dff_load1 = dff (or2 st_load0 (and2 dff_load1 io_DMA))
st_load1  = and2 cpu dff_load1
dff_load2 = dff (or2 st_load1 (and2 dff_load2 io_DMA))
st_load2  = and2 cpu dff_load2
```

Suppose `io_DNA` is 0 for a number of clock cycles, and the machine is dispatching an instruction with secondary opcide 1. That indicates a `load` instruction, and . Consider what happens during a sequence of clock cycles.

- Suppose that during cycle 100 `pRX!!1` is 1.
- Cycle 101
  - At the clock tick beginning the cycle, `dff_load0` will become 1
  - Suppose that during cycle 101, `io_DMA` is 1, so `cpu` is 1. Since `dff_load0` is 1, `st_load0` is also 1, and the control signals for the `load0` state will all be 1. The processor will perform the first step of the `load` instruction. What that really means is that any flip flop changes required will occur st the clock tick that ends Cycle 101 and begins cycle 102.

- Cycle 102
    - At the clock tick between cycles 101 and 102, `dff_load1` becomes 1 and `dff_load0` becomes 0.
    - But suppose that the I/O system needs to steal the cycle to do its own memory access. To do this, the system sets `io_DNA` = 1m and that makes `cpu` = 0.
    - Although `dff_load1` = 1, the corresponding signal `st_load1` is 0 during this cycle. This is because `st_load1 = and2 cpu dff_load1` and the value of `cpu` is 0. Consequently, the control signals set byy `st_load1` all remain 0. Any register updates belonging to `st=load1` will not happen at the next clock tick. Notice that all the cominbational logic calculations for `st_load1` will go ahead; but their results will not be latched into any register at the clock tick. These logic calculations will be repeated during the next clock cycle.
    - Because the processor will not update any registers at the end of the cycle, it is safe for the I/O to set the controls for the memory that it needs, as well as the memory address and data words.
    - At the clock tick ending this cycle, the I/O memory access takes place and the processor does nothing: its cycle has been stolen.
-

6.3 The control circuit

The Control module presents the control algorithm using imperative programming language syntax in comments, and then defines the control circuit which is derived from the algorithm.

```
-- Sigma16 M1: ALU.hs
-- John T. O'Donnell, 2021
-- See Sigma16/README and https://jtod.github.io/home/Sigma16/

-- Control Algorithm and control circuit

{-# LANGUAGE NamedFieldPuns #-}
{-# LANGUAGE RecordWildCards #-}

module Circuit.Control where
```

```
import HDL.Hydra.Core.Lib
import HDL.Hydra.Circuits.Combinational
import Circuit.Interface

-- This is the high level control algorithm, written using
-- assignment statements to describe the effect that will take
-- place at the end of a clock cycle.  Each statement is decorated
-- with the list of control signals that must be asserted during
-- the clock cycle to make the datapath perform the operation.
-- Some of the Sigma16 instructions are unimplemented, but the
-- key ones are all defined.
```

The control algorithm attaches a label to each state. The state labels begin with `st_` and describe the specific state. For example, `st_instr_fet` is the state where the next instruction is fetched. Following the state label is a statement that describes the intended effect of the state, followed by the control signal settings required to achieve the effect.

The statements use ordinary imperative programming language syntax because of its familiarity. However, these statements are not executed directly; they are only comments that describe what's going on. The actualy implementation of the algorithm is achieved using flip flops and logic gates, which are defined later.

This version of M1 leaves the mul and div instructions unimplemented.

```
{-
repeat forever
  st_instr_fet:
    ir := mem[pc], pc++;
       {ctl_ma_pc, ctl_ir_ld, ctl_x_pc, ctl_alu=alu_inc, ctl_pc_ld}
  st_dispatch:
  case ir_op of

    0 -> -- add instruction
        st_add:  reg[ir_d] := reg[ir_sa] + reg[ir_sb]
          assert [ctl_alu_abc=000, ctl_rf_alu, ctl_rf_ld, ctl_rf_ldcc]

    1 -> -- sub instruction
        st_sub:  reg[ir_d] := reg[ir_sa] - reg[ir_sb]
          assert [ctl_alu_abc=001, ctl_rf_alu, ctl_rf_ld, ctl_rf_ldcc]
```

```
2 -> -- mul instruction
    -- unimplemented, does nothing

3 -> -- div instruction
    -- unimplemented, does nothing

4 -> -- cmp instruction
    st_cmp:  reg[15] := alu_cmp (reg[ir_sa], reg[ir_sb])
       assert [ctl_alu_abc=100, ctl_rf_ldcc]

11 -> -- trap instruction
    st_trap0:
      -- The simulation driver detects that a trap has been
      -- executed by observing when the control algorithm
      -- enters this state, and it performs the system action
      -- requested by the trap operand in Rd.

15 -> -- expand to RX format
    -- This code allows expansion to a two-word format with
    -- an address operand.  The instruction is determined by a
    -- secondary opcode in the b field of the ir

  case ir_sb of

    0 -> -- lea instruction
        st_lea0:  ad := mem[pc], pc++
          assert [ctl_ma_pc, ctl_ad_ld, ctl_x_pc, ctl_alu_abc=011, ctl_pc_ld]
        st_lea1:  reg[ir_d] := reg[ir_sa] + ad
          assert [ctl_y_ad, ctl_alu=alu_add, ctl_rf_alu, ctl_rf_ld]

    1 -> -- load instruction
        st_load0:  ad := mem[pc], pc++
          assert [ctl_ma_pc, ctl_ad_ld, ctl_x_pc, ctl_alu_abc=011, ctl_pc_ld]
        st_load1:  ad := reg[ir_sa] + ad
          assert [ctl_y_ad, ctl_alu_abc=000, ctl_ad_ld, ctl_ad_alu]
        st_load2:  reg[ir_d] := mem[ad]
            assert [ctl_rf_ld]

    2 -> -- store instruction
        st_store0:  ad := mem[pc], pc++
          assert [ctl_ma_pc, ctl_ad_ld, ctl_x_pc, ctl_alu_abc=011, ctl_pc_ld]
        st_store1:  ad := reg[ir_sa] + ad
          assert [ctl_y_ad, ctl_alu_abc=000, ctl_ad_ld, ctl_ad_alu]
        st_store2:
          mem[addr] := reg[ir_d]
            assert [ctl_rf_sd, ctl_sto]

    3 -> --  jump instruction
        st_jump0:  ad := mem[pc], pc++
          assert [ctl_ma_pc, ctl_ad_ld, ctl_x_pc, ctl_alu_abc=011, ctl_pc_ld]
        st_jump1:  ad := reg[ir_sa] + ad
          assert [ctl_y_ad, ctl_alu_abc=000, ctl_ad_ld, ctl_ad_alu]
        st_jump2:  pc := ad
          assert [ctl_pc_ad, ctl_pc_ld]

    4 -> -- jumpc0 instruction
        st_jumpc00:  ad := mem[pc], pc++
          assert [ctl_ma_pc, ctl_ad_ld, ctl_x_pc, ctl_alu_abc=011, ctl_pc_ld]
        st_jumpc01:  ad := reg[ir_sa] + ad
          assert [ctl_y_ad, ctl_alu_abc=000, ctl_ad_ld, ctl_ad_alu]
        st_jumpc02:  if inv condcc then pc := ad
          assert [ctl_pc_ad, if inv condcc then ctl_pc_ld]

    5 -> -- jumpc1 instruction
        st_jumpc10:  ad := mem[pc], pc++
          assert [ctl_ma_pc, ctl_ad_ld, ctl_x_pc, ctl_alu_abc=011, ctl_pc_ld]
        st_jumpc11:  ad := reg[ir_sa] + ad
          assert [ctl_y_ad, ctl_alu_abc=000, ctl_ad_ld, ctl_ad_alu]
        st_jumpc12: if condcc then pc := ad
          assert [ctl_pc_ad, if condcc then ctl_pc_ld]

    6 -> -- jal instruction
        st_jal0:  ad := mem[pc], pc++
          assert [ctl_ma_pc, ctl_ad_ld, ctl_x_pc, ctl_alu_abc=011, ctl_pc_ld]
        st_jal1:  ad := reg[ir_sa] + ad
          assert [ctl_y_ad, ctl_alu_abc=000, ctl_ad_ld, ctl_ad_alu]
        st_jal2: reg[ir_d] := pc, pc := ad,
          assert [ctl_rf_ld, ctl_rf_pc, ctl_pc_ld, ctl_pc_ad]
```

```
-- The remaining opcodes are used in the full Sigma16 architecture,
-- but in the Core they are unimplemented and treated as nop

        7 -> -- nop
        8 -> -- nop
        9 -> -- nop
        10 -> -- nop
        11 -> -- nop
        12 -> -- nop
        13 -> -- nop
        14 -> -- nop
        15 -> -- nop
-}
```

The control circuit takes the following inputs:

- `reset` (1 bit) If `reset` is 1 during a clock cycle, the circuit will enter its initial state at the next clock tick.

```
----------------------------------------------------------------------
--   Control circuit
----------------------------------------------------------------------

control
  :: CBit a
  => a         -- reset
  -> [a]       -- ir
  -> [a]       -- cc
  -> SysIO a   -- I/O
  -> (CtlState a, a, CtlSig a)

control reset ir cc  (SysIO {..}) = (ctlstate,start,ctlsigs)
  where

-- Fields of instruction and conditional control
      ir_op = field ir  0 4       -- instruction opcode
      ir_d  = field ir  4 4       -- instruction destination register
      ir_sa = field ir  8 4       -- instruction source a register
      ir_sb = field ir 12 4       -- instruction source b register
      condcc = indexbit ir_d cc
```

```
-- Control mode is either io_DMA or cpu
      cpu = inv io_DMA

-- Control states
      start = orw
        [reset,
         st_add, st_sub, st_mul0, st_cmp, st_trap0,
         st_lea1,  st_load2, st_store2, st_jump2,
         st_jumpc02, st_jumpc12, st_jal2]
      st_start = and2 start cpu

      dff_instr_fet = dff (or2 st_start (and2 dff_instr_fet io_DMA))
      st_instr_fet  = and2 dff_instr_fet cpu

      dff_dispatch = dff (or2 st_instr_fet (and2 dff_dispatch io_DMA))
      st_dispatch  = and2 dff_dispatch cpu
      pRRR = demux4w ir_op st_dispatch
      pXX  = demux4w ir_sb (pRRR!!14)
      pRX  = demux4w ir_sb (pRRR!!15)

-- lea control states
      dff_lea0 = dff (or2 (pRX!!0) (and2 dff_lea0 io_DMA))
      st_lea0  = and2 dff_lea0 cpu
      dff_lea1 = dff (or2 st_lea0 (and2 dff_lea1 io_DMA))
      st_lea1  = and2 cpu dff_lea1
      dff_lea2 = dff (or2 st_lea1 (and2 dff_lea2 io_DMA))
      st_lea2  = and2 dff_lea2 cpu

-- load control states
      dff_load0 = dff (or2 (pRX!!1) (and2 dff_load0 io_DMA))
      st_load0  = and2 dff_load0 cpu
      dff_load1 = dff (or2 st_load0 (and2 dff_load1 io_DMA))
      st_load1  = and2 cpu dff_load1
      dff_load2 = dff (or2 st_load1 (and2 dff_load2 io_DMA))
      st_load2  = and2 cpu dff_load2

-- store control states
      dff_store0 = dff (or2 (pRX!!2) (and2 dff_store0 io_DMA))
      st_store0  = and2 cpu dff_store0
      dff_store1 = dff (or2 st_store0 (and2 dff_store1 io_DMA))
      st_store1  = and2 dff_store1 cpu
      dff_store2 = dff (or2 st_store1 (and2 dff_store2 io_DMA))
      st_store2  = and2 dff_store2 cpu

-- jump control states
      dff_jump0 = dff (or2 (pRX!!3) (and2 dff_jump0 io_DMA))
      st_jump0  = and2 dff_jump0 cpu
      dff_jump1 = dff (or2 st_jump0 (and2 dff_jump1 io_DMA))
      st_jump1  = and2 dff_jump1 cpu
      dff_jump2 = dff (or2 st_jump1 (and2 dff_jump2 io_DMA))
      st_jump2  = and2 cpu dff_jump2

-- jumpc0 control states
      dff_jumpc00 = dff (or2 (pRX!!4) (and2 dff_jumpc00 io_DMA))
      st_jumpc00  = and2 dff_jumpc00 cpu
      dff_jumpc01 = dff (or2 st_jumpc00 (and2 dff_jumpc01 io_DMA))
      st_jumpc01  = and2 dff_jumpc01 cpu
      dff_jumpc02 = dff (or2 st_jumpc01 (and2 dff_jumpc02 io_DMA))
      st_jumpc02  = and2 dff_jumpc02 cpu

-- jumpc1 control states
      dff_jumpc10 = dff (or2 (pRX!!5) (and2 dff_jumpc10 io_DMA))
      st_jumpc10  = and2 dff_jumpc10 cpu
      dff_jumpc11 = dff (or2 st_jumpc10 (and2 dff_jumpc11 io_DMA))
      st_jumpc11  = and2 dff_jumpc11 cpu
      dff_jumpc12 = dff (or2 st_jumpc11 (and2 dff_jumpc12 io_DMA))
      st_jumpc12  = and2 dff_jumpc12 cpu

-- jal control states
      dff_jal0 = dff (or2 (pRX!!6) (and2 dff_jal0 io_DMA))
      st_jal0  = and2 dff_jal0 cpu
      dff_jal1 = dff (or2 st_jal0 (and2 dff_jal1 io_DMA))
      st_jal1  = and2 dff_jal1 cpu
      dff_jal2 = dff (or2 st_jal1 (and2 dff_jal2 io_DMA))
      st_jal2  = and2 dff_jal2 cpu

-- RRR control states
      dff_add   = dff (or2 (pRRR!!0) (and2 dff_add io_DMA))
      st_add    = and2 dff_add cpu
      dff_sub   = dff (or2 (pRRR!!1) (and2 dff_sub io_DMA))
      st_sub    = and2 dff_sub cpu

      dff_mul0  = dff (or2 (pRRR!!2) (and2 dff_mul0 io_DMA))
      st_mul0   = and2 dff_mul0 cpu

      dff_div0  = dff (or2 (pRRR!!3) (and2 dff_div0 io_DMA))
      st_div0   = and2 dff_div0 cpu

      dff_cmp   = dff (or2 (pRRR!!4) (and2 dff_cmp io_DMA))
      st_cmp    = and2 dff_cmp cpu

      dff_trap0 = dff (or2 (pRRR!!11) (and2 dff_trap0 io_DMA))
      st_trap0  = and2 dff_trap0 cpu

-- Generate control signals
      ctl_rf_ld   = orw [st_load2,st_lea1,st_add,st_sub,
                           st_jal2]
      ctl_rf_ldcc = orw [st_cmp, st_add, st_sub]
      ctl_rf_pc   = orw [st_jal2]
      ctl_rf_alu  = orw [st_lea1,st_add,st_sub]
      ctl_rf_sd   = orw [st_store2,st_jumpc00]
      ctl_alu_a   = orw [st_cmp]
      ctl_alu_b   = orw [st_instr_fet,st_load0,st_store0,st_lea0,
                         st_jump0, st_jumpc00, st_jumpc10, st_jal0]
      ctl_alu_c   = orw [st_instr_fet,st_load0,st_store0,st_lea0,
                         st_jump0, st_jumpc00, st_jumpc10,
                         st_sub,st_jumpc00,st_jal0]
      ctl_ir_ld   = orw [st_instr_fet]
      ctl_pc_ld   = orw [st_instr_fet, st_lea0, st_load0, st_store0,
                           st_jump0, st_jump2,
                           st_jumpc00, and2 (inv condcc) st_jumpc02,
                           st_jumpc10, and2 condcc st_jumpc12,
                           st_jal0, st_jal2]
      ctl_pc_ad   = orw [st_jump2, st_jumpc02, st_jumpc12, st_jal2]
      ctl_ad_ld   = orw [st_load0,st_load1,st_lea0,st_store0,
                         st_store1,st_jumpc00,st_jumpc01,
                         st_jumpc10,st_jumpc11,st_jump0,st_jump1,
                         st_jal0,st_jal1]
      ctl_ad_alu  = orw [st_load1,st_store1,st_jump1,st_jumpc01,st_jumpc11,st_jal1]
      ctl_ma_pc   = orw [st_instr_fet,st_load0,st_lea0,st_store0,
                         st_jumpc10,st_jumpc00,st_jump0,st_jal0]
      ctl_x_pc    = orw [st_instr_fet,st_load0,st_lea0,st_store0,
                         st_jumpc10,st_jumpc00,st_jump0,st_jal0]
      ctl_y_ad    = orw [st_load1,st_store1,st_lea1,st_jumpc11,
                       st_jumpc01,st_jump1,st_jal1]
      ctl_sto     = orw [st_store2]

-- Record of control states and signals
      ctlstate = CtlState {..}
      ctlsigs = CtlSig {..}

indexbit :: Bit a => [a] -> [a] -> a
indexbit [] [x] = x
indexbit (c:cs) xs =
  mux1 c (indexbit cs (drop i xs))
         (indexbit cs (take i xs))
  where i = 2 ^ length cs
```

7 Memory

```
-- Sigma16: Memory.hs
-- John T. O'Donnell, 2021
-- See Sigma16/README and https://jtod.github.io/home/Sigma16/

------------------------------------------------------------------------
-- Memory
------------------------------------------------------------------------

module Circuit.Memory where

import HDL.Hydra.Core.Lib
import HDL.Hydra.Circuits.Combinational
import HDL.Hydra.Circuits.Register

-- Defines memory circuits using flip flops and logic gates

-- memw generates a memory circuit with 2^k n-bit words, using the two
-- size parameters n and k.
```

```
memw
  :: CBit a
  => Int    -- n = wordsize
  -> Int    -- k = real address size: 2^k words in memory
  -> a      -- sto control
  -> [a]    -- p = address
  -> [a]    -- x = data to store. if sto then mem[p]:=x
  -> [a]    -- y = data fetched = mem[p]

memw n k sto p x = mapn (mem1 k sto p) n x


-- mem1 generates a memory circuit with 2^k bits, using the size
-- parameter n.

mem1
  :: CBit a
  => Int    -- k = real address size: 2^k words in memory
  -> a      -- sto control
  -> [a]    -- p = address
  -> a      -- x = data to store. if sto then mem[p]:=x
  -> a      -- y = data fetched = mem[p]

mem1 k sto p x
  | k==0  =  reg1 sto x
  | k>0   =  mux1 q m0 m1
  where
    (q:qs) = p
    (sto0,sto1) = demux1 q sto
    m0 = mem1 (k-1) sto0 qs x
    m1 = mem1 (k-1) sto1 qs x
```

8 System

I/O control inputs

dma	1 bit	indicates stolen clock cycle	
`dma_store`	1 bit	mem[dma_a] := dma_d	
`dma_fetch`	1 bit	m_{out} = mem[dma_a]	
dma_{reg}	1 bit	x = reg[dma_a] (least significant 4 bits	
dma_a	16 bits	address	
dma_d	16 bits	data	
`=a_b=`	`a_b`	`a_b`	`~a_b~`

```
-- Sigma16: M1.hs
-- John T. O'Donnell, 2021
-- See Sigma16/README and https://jtod.github.io/home/Sigma16/

{-# LANGUAGE NamedFieldPuns #-}
{-# LANGUAGE RecordWildCards #-}

module Circuit.System
  ( m1
  , module Circuit.Interface
  , module Circuit.ALU
  , module Circuit.Datapath
  , module Circuit.Control
  , module Circuit.Memory

  ) where

import HDL.Hydra.Core.Lib
import HDL.Hydra.Circuits.Combinational
import HDL.Hydra.Circuits.Register

import Circuit.Interface
import Circuit.ALU
import Circuit.Datapath
import Circuit.Control
import Circuit.Memory

-- M1 is a digital circuit that implements the Core subset of the
-- Sigma16 instruction set architecture, apart from mul and div, which
-- are unimplemented.
```

```
------------------------------------------------------------------------
{- Instruction set architecture of Sigma16

RRR instructions
------------------------------------------------------------------------

 op   format  mnemonic   operands   action
---- -------- ---------- ---------- ------------------------------------
 0     RRR    add        R1,R2,R3   R1 := R2+R3
 1     RRR    sub        R1,R2,R3   R1 := R2-R3
 2     RRR    mul        R1,R2,R3   R1 := R2*R3, R15 := high word
 3     RRR    div        R1,R2,R3   R1 := R2/R3, R15 := R2 mod R3
 4     RRR    cmp        R2,R3      R15 := R2 cmp R3
 b     RRR    trap       R1,R2,R3   trap interrupt
 e     EXP                          (expand to EXP format)
 f     RX                           (expand to RX format)

Table: **Instructions represented in RRR format**
------------------------------------------------------------------------

RX instructions
------------------------------------------------------------------------

 op   b   format  mnemonic   operands   action
---- --- -------- ---------- ---------- --------------------------------
 f    0     RX    lea        Rd,x[Ra]   Rd := x+Ra
 f    1     RX    load       Rd,x[Ra]   Rd := mem[x+Ra]
 f    2     RX    store      Rd,x[Ra]   mem[x+Ra] := Rd
 f    3     RX    jump       x[Ra]      pc := x+Ra
 f    4     RX    jumpc0     Rd,x[Ra]   if Rd==0 then pc := x+Ra
 f    5     RX    jumpc1     Rd,x[Ra]   if Rd/=0 then pc := x+Ra
 f    6     RX    jal        Rd,x[Ra]   Rd := pc, pc := x+Ra

Table: **Instructions represented by RX and X formats**
------------------------------------------------------------------------

I/O control inputs
dma           1 bit    indicates stolen clock cycle
dma_store     1 bit    mem[dma_a] := dma_d
dma_fetch     1 bit    m_out = mem[dma_a]
```

```
dma_reg       1 bit    x = reg[dma_a]  (least significant 4 bits
dma_a         16 bits  address
dma_d         16 bits  data
-}

m1 reset (SysIO {..}) =
  (ctl_state, ctl_start, ctlsigs,              -- control
   dp,                                         -- datapath
   m_sto, m_addr, m_real_addr, m_data, m_out)  -- memory
  where

-- Size parameters
    n = 16        -- word size is n, and address space is 2^n words
    msize = 16    -- installed memory contains 2^msize words
      -- if msize=n then full memory is available
      -- if msize<n the simulation may be faster but prog has less memory

-- Datapath
    dp = datapath ctlsigs (SysIO {..}) m_out
    (r,ccnew) = aluOutputs dp

-- Control
    (ctl_state, ctl_start, ctlsigs) =
             control reset (ir dp) (cc dp) (SysIO {..})

-- Memory
    m_real_addr = field m_addr (n-msize) msize
    m_out = memw n msize m_sto m_real_addr m_data

-- Input/Output using DMA
    m_sto = or2 (and2 io_DMA io_memStore)               -- I/O store
                (and2 (inv io_DMA) (ctl_sto ctlsigs))  -- CPU store
    m_data = mux1w io_DMA (md dp) io_data
    m_addr = mux1w io_DMA (ma dp) io_address
```

9 Reading the simulation output

9.1 Starting a simulation

The `:main` command starts the simulation driver. Its operand is the object file that will be executed. Just give the base name of the file (`Add`), not the full name (`Add.obj.txt`).

The driver reads the object code file and siaplays the object code, which is a list of numbers. Then it generates the inputs to the circuit that will be required to boot the program. This is a list of strings; each string gives the input signal values for one clock cycle. These signals tell the circuit to perform an I/O operation and that the operation is an external read into memory. Furthermore, the memory address to use and the data value are specified. For example, the first element of the list gives the inputs for clock cycle 0, and the last two numbers mean that the input will store into address 0 and the value to store is 61697.

```
ghci> :main programs/Add
Sigma16 M1 system starting
Reading object file programs/Add.obj.txt
Object code is [61697,8,61953,9,786,62210,10,45056,23,14,0]
Boot system inputs = ["0 1 1 0 0 0 61697","0 1 1 0 0 1 8",
"0 1 1 0 0 2 61953","0 1 1 0 0 3 9","0 1 1 0 0 4 786",
"0 1 1 0 0 5 62210","0 1 1 0 0 6 10","0 1 1 0 0 7 45056",
"0 1 1 0 0 8 23","0 1 1 0 0 9 14","0 1 1 0 0 10 0"]
M1>
```

10 M1 simulation driver: Run

```
-- Sigma16 M1.Run, Simulation driver for M1 circuit implementation of Sigma16
-- John T. O'Donnell, 2021
-- See Sigma16/README and https://jtod.github.io/home/Sigma16/

-- Usage:
--   ghci                 -- start ghci and initialize using .ghci
--   :load Run            -- run M1 circuit on examples/Core/Simple/Add.obj.txt
--   :main programs/Add   -- run M1 circuit on examples/Core/Simple/Add.obj.txt
--   run                  -- run the Add program on the circuit
--   help                 -- list the M1 simulation driver commands
--   quit                 -- quit ghci, return to shell
```

```
{-# LANGUAGE NamedFieldPuns #-}
{-# LANGUAGE RecordWildCards #-}

module M1.Run where

import HDL.Hydra.Core.Lib   -- Hydra hardware description language
import ReadObj              -- read object code file
import Circuit.System       -- the M1 circuit

import System.Environment
import System.IO
import Control.Monad.State
import Control.Exception
import qualified Data.Map as Map

--------------------------------------------------------------------------------
-- M1 simulation driver
--------------------------------------------------------------------------------

main :: IO ()
main = driver $ do
  printLine "Sigma16 M1 system starting"
  objectCode <- liftIO getObject
  let bootData = bootInputs objectCode
  putStoredInput bootData
  printLine ("Boot system inputs = " ++ show bootData)

  -- Input ports
  in_reset       <- inPortBit  "reset"
  in_io_DMA      <- inPortBit  "io_DMA"
  in_io_memStore  <- inPortBit  "io_memStore"
  in_io_memFetch  <- inPortBit  "io_memFetch"
  in_io_regFetch  <- inPortBit  "io_regFetch"
  in_io_address   <- inPortWord "io_address" 16
  in_io_data      <- inPortWord "io_data" 16

  -- Input signals
  let reset         = inbsig in_reset
  let io_DMA        = inbsig in_io_DMA
  let io_memStore   = inbsig in_io_memStore
  let io_memFetch   = inbsig in_io_memFetch
  let io_regFetch   = inbsig in_io_regFetch
  let io_address    = inwsig in_io_address
  let io_data       = inwsig in_io_data
  let io = SysIO {..}

-- The M1 circuit
  let  (CtlState {..}, ctl_start, (CtlSig {..}), dp,
        m_sto, m_addr, m_real_addr, m_data, m_out)
         = m1 reset io

-- Prepare for memory and register dump
  setPeek m_out
  setPeek (b dp)

-- Prepare for breakpoints
  let ctlStateLookupTable =
        [ ("reset", reset)
        , ("st_instr_fet",  st_instr_fet)
        , ("st_dispatch",   st_dispatch)
        , ("st_add",        st_add)
        , ("st_sub",        st_sub)
        , ("st_mul0",       st_mul0)
        , ("st_div0",       st_div0)
        , ("st_cmp",        st_cmp)
        , ("st_trap0",      st_trap0)
        , ("st_lea0",       st_lea0)
        , ("st_load0",      st_load0)
        , ("st_store0",     st_store0)
        , ("st_jump0",      st_jump0)
        , ("st_jumpc00",    st_jumpc00)
        , ("st_jumpc10",    st_jumpc10)
        , ("st_jal0",       st_jal0)
        ]
  let flags = ctlStateLookupTable
  setFlagTable flags

-- Define names for subsystem outputs
  let (r,ccnew) = aluOutputs dp
```

```
-- Format the output
  format
    [ string "\nSystem control\n"
    , string "  reset = ", bit reset
    , string "  cpu = ", bit cpu
    , string "  ctl_start = ", bit ctl_start
    , string "\n"
    , string "\nInput/Output\n"
    , string "  io_DMA = ", bit io_DMA
    , string "  io_memStore = ", bit io_memStore
    , string "  io_memFetch = ", bit io_memFetch
    , string "  io_regFetch = ", bit io_regFetch
    , string "\n"
    , string "  io_address = ", binhex io_address
    , string "  io_data = ", binhex io_data
    , string "\n"
    , string "\nControl state\n  "
    , string " st_instr_fet = ", bit dff_instr_fet, bit st_instr_fet
    , string "  st_dispatch = ", bit dff_dispatch, bit st_dispatch
    , string "       st_add = ", bit dff_add, bit st_add
    , string "       st_sub = ", bit dff_sub, bit st_sub
    , string "\n  "
    , string "      st_mul0 = ", bit dff_mul0, bit st_mul0
    , string "      st_div0 = ", bit dff_div0, bit st_div0
    , string "       st_cmp = ", bit dff_cmp, bit st_cmp
    , string "     st_trap0 = ", bit dff_trap0, bit st_trap0
    , string "\n  "
    , string "      st_lea0 = ", bit dff_lea0, bit st_lea0
    , string "      st_lea1 = ", bit dff_lea1, bit st_lea1
    , string "      st_lea2 = ", bit dff_lea2, bit st_lea2
    , string "     st_load0 = ", bit dff_load0, bit st_load0
    , string "\n  "
    , string "     st_load1 = ", bit dff_load1, bit st_load1
    , string "     st_load2 = ", bit dff_load2, bit st_load2
    , string "    st_store0 = ", bit dff_store0, bit st_store0
    , string "    st_store1 = ", bit dff_store1, bit st_store1
    , string "\n  "
    , string "    st_store2 = ", bit dff_store2, bit st_store2
    , string "     st_jump0 = ", bit dff_jump0, bit st_jump0
    , string "     st_jump1 = ", bit dff_jump1, bit st_jump1
    , string "     st_jump2 = ", bit dff_jump2, bit st_jump2
    , string "\n  "
    , string "   st_jumpc00 = ", bit dff_jumpc00, bit st_jumpc00
    , string "   st_jumpc01 = ", bit dff_jumpc01, bit st_jumpc01
    , string "   st_jumpc02 = ", bit dff_jumpc02, bit st_jumpc02
    , string "   st_jumpc10 = ", bit dff_jumpc10, bit st_jumpc10
    , string "\n  "
    , string "   st_jumpc11 = ", bit dff_jumpc11, bit st_jumpc11
    , string "   st_jumpc12 = ", bit dff_jumpc12, bit st_jumpc12
    , string "      st_jal0 = ", bit dff_jal0, bit st_jal0
    , string "      st_jal1 = ", bit dff_jal1, bit st_jal1
    , string "\n  "
    , string "      st_jal2 = ", bit dff_jal2, bit st_jal2

    , string "\n\nControl signals\n  "
    , string "    ctl_alu_a = ", bit ctl_alu_a
    , string "    ctl_alu_b = ", bit ctl_alu_b
    , string "    ctl_alu_c = ", bit ctl_alu_c
    , string "     ctl_x_pc = ", bit ctl_x_pc
    , string "\n  "
    , string "     ctl_y_ad = ", bit ctl_y_ad
    , string "    ctl_rf_ld = ", bit ctl_rf_ld
    , string "  ctl_rf_ldcc = ", bit ctl_rf_ldcc
    , string "    ctl_rf_pc = ", bit ctl_rf_pc
    , string "\n  "
    , string "    ctl_pc_ld = ", bit ctl_pc_ad
    , string "    ctl_pc_ad = ", bit ctl_pc_ad
    , string "   ctl_rf_alu = ", bit ctl_rf_alu
    , string "    ctl_rf_sd = ", bit ctl_rf_sd
    , string "\n  "
    , string "    ctl_ir_ld = ", bit ctl_ir_ld
    , string "    ctl_pc_ld = ", bit ctl_pc_ld
    , string "    ctl_ad_ld = ", bit ctl_ad_ld
    , string "   ctl_ad_alu = ", bit ctl_ad_alu
    , string "\n  "
    , string "    ctl_ma_pc = ", bit ctl_ma_pc
    , string "      ctl_sto = ", bit ctl_sto

    , string "\n\nALU\n"
```

```
    , string "  ALU inputs: "
    , string "  operation = ", bit ctl_alu_a, bit ctl_alu_b, bit ctl_alu_c
    , string "  x = ", binhex (x dp)
    , string "  y = ", binhex (y dp)
    , string "  cc = ", binhex (cc dp)
    , string "  ir_d = ", binhex (ir_d dp)
    , string "\n  ALU outputs: "
    , string "  r = ", binhex r
    , string "  ccnew = ", binhex ccnew
    , string "  condcc = ", bit condcc

    , string "\n\nDatapath\n  "
    , string "    ir = ", binhex (ir dp)
    , string "    pc = ", binhex (pc dp)
    , string "    ad = ", binhex (ad dp)
    , string "    cc = ", binhex (cc dp)
    , string "\n  "
    , string "     a = ", binhex (a dp)
    , string "     b = ", binhex (b dp)
    , string "     x = ", binhex (x dp)
    , string "     y = ", binhex (y dp)
    , string "\n  "
    , string "     p = ", binhex (p dp)
    , string "     q = ", binhex (q dp)
    , string "     r = ", binhex (r)
    , string "\n  "
    , string "    ma = ", binhex (ma dp)
    , string "    md = ", binhex (md dp)

-- Memory interface
    , string "\n\nMemory\n  "
    , string "  m_sto = ", bit m_sto
    , string "  m_addr = ", binhex m_addr
    , string "  m_real_addr = ", binhex m_real_addr
    , string "  m_data = ", binhex m_data
    , string "  m_out =", binhex m_out
    , string "\n"

-- ...................................................................
-- Higher level analysis of what happened on this cycle.  The
-- following actions examine various signals in order to detect what
-- is happening in the machine, and they print higher level
-- description.
-- ...........................................................

-- Print a message when the system is reset

    ,  fmtIf reset
           [string ("\n" ++ take 72 (repeat '*') ++ "\n"),
            string "Reset: control algorithm starting",
            string ("\n" ++ take 72 (repeat '*'))]
           [],

-- When the displacement for an RX instruction is fetched, save
-- it in the simulation driver state

         fmtIf (orw [st_lea1, st_load1, st_store1, st_jump1, st_jumpc01,
                     st_jumpc11, st_jal1])
           [setStateWs setDisplacement [(ad dp)],
            string "*** Fetched displacement = ",
            simstate showDisplacement
           ]
           [],

-- Record the effective address when it is calculated.  This is the r
-- output of the ALU, and usually will be loaded into the ad register.

         fmtIf (orw [st_lea1, st_load1, st_store1, st_jump1,
                     st_jumpc01, st_jumpc11, st_jal1])
         [setStateWs setEffAddr [r]]
           [],

-- Process a load to the register file
         fmtIf ctl_rf_ld
           [string "Register file update: ",
            string "R",
            bindec 1 (field (ir dp) 4 4),
            string " := ", hex (p dp),
--            setStateWs setRfLoad [field (ir dp) 4 4, (p dp)],
            setStateWsIO setRfLoad [field (ir dp) 4 4, (p dp)],
            string "\n"
            ]
           [],

-- Process a store to memory
         fmtIf ctl_sto
           [string "Memory store:  ",
            string "mem[",
            binhex m_addr,
            string "] := ", hex m_data,
            setStateWsIO setMemStore [m_addr, m_data]
           ]
           [],

-- If an instruction was completed during this clock cycle, fetch it,
-- decode it, and print it.  The first word of the instruction is in
-- ir, the second word (if it is an RX instruction) is in the
-- displacement field of the simulation driver state.

         fmtIf (and2 ctl_start (inv reset))
           [string ("\n" ++ take 72 (repeat '*') ++ "\n"),
            string "Executed instruction:  ",
            fmtWordsGeneral findMnemonic [field (ir dp) 0 4, field (ir dp) 12 4],
            string " ",
            fmtIf (orw [st_add, st_sub, st_cmp])

              [string " R", bindec 1 (field (ir dp) 4 4),    -- RRR format
               string ",R", bindec 1 (field (ir dp) 8 4),
               string ",R", bindec 1 (field (ir dp) 12 4)]
              [string " R", bindec 1 (field (ir dp) 4 4),    -- RX format
               string ",",
               simstate showDisplacement,
               string "[R", bindec 1 (field (ir dp) 8 4), string "]" ], -- ,
--               string "   effective address = ",
--               simstate showEffAddr],
            string "\n",
            simstate showRfLoads,
            setStateWs clearRfLoads [],
            simstate showMemStores,
            setStateWs clearMemStores [],
 --- Describe effect of jumps
            fmtIf st_jumpc02
              [fmtIf condcc
                 [string "jumpc0 instruction will not jump\n"]
                 [string "jumpc0 instruction is jumping to ",
                  binhex (ad dp), string "\n"]]
              [],

            fmtIf st_jumpc12
               [fmtIf condcc
                 [string "jumpc1 instruction is jumping to ",
                  binhex (ad dp), string "\n"]
                 [string "jumpc1 instruction will not jump\n"]]
               [],

            fmtIf st_jump2
              [string "jump instruction is jumping to ",
               binhex (ad dp), string "\n"]
              [],
            fmtIf st_jal2
              [string "jal instruction is jumping to ",
               binhex (ad dp), string "\n"]
              [],

-- Display instruction control registers
            string "Processor state:  ",
            string "  pc = ", binhex (pc dp),
            string "  ir = ", binhex (ir dp),
            string "  ad = ", binhex (ad dp),
            string ("\n" ++ take 72 (repeat '*'))
              ]
           [],

-- If a trap is being executed, indicate this in the simulation driver
-- state, so the driver can terminate the simulation

         fmtIf st_trap0
           [setStateWs setTrap [],
            setHalted,
            string ("\n" ++ take 72 (repeat '*') ++ "\n"),
            string "System trap request:  Halt\n",
            string "Processor has halted\n",
            string (take 72 (repeat '*') ++ "\n")
           ]
           []
    ]

-- This ends definitions of the tools; the driver algorithms starts
-- now
  startup
  printLine "M1 Run finished"

--------------------------------------------------------------------------------
-- Booter
--------------------------------------------------------------------------------

getObject :: IO [Int]
getObject = do
  args <- getArgs
--  putStrLn (show args)
  case args of
    [] -> do
      putStrLn "Usage: :main path/to/objectfile"
      return []
    (a:_) -> do
      prefixFile <- maybeRead "fileprefix.txt"
--      putStrLn ("file prefix = " ++ show prefixFile)
      let basePath =
            case prefixFile of
              Nothing -> a
              Just p -> lines p !! 0 ++ a
      let fullPath = basePath ++ ".obj.txt"
      putStrLn ("Reading object file " ++ fullPath)
      code <- liftIO $ readObjectCode fullPath
      putStrLn ("Object code is " ++ show code)
      return code

-- Generate control signals to boot object code
bootInputs :: [Int] -> [String]
bootInputs code =
--  code <- readObjectFile
  let f i x = "0 1 1 0 0 " ++ show i ++ " " ++ show x
      inps = zipWith f [0..] code
  in inps

putStoredInput :: [String] -> StateT (SysState a) IO ()
putStoredInput storedInput = do
  s <- get
  put $ s {storedInput}

--------------------------------------------------------------------------------
-- Top level M1 control
--------------------------------------------------------------------------------

conditional :: Bool -> StateT (SysState a) IO ()
  -> StateT (SysState a) IO ()
conditional b op =
  case b of
    True -> do op
               return ()
    False -> return ()

peekReg :: Int -> StateT (SysState DriverState) IO ()
peekReg regnum = do
  s <- get
  let displayFullSignals = False
  let i = cycleCount s
  let inp = inPortList s
  let outp = outPortList s
  conditional displayFullSignals $ do
    liftIO $ putStrLn (take 80 (repeat '-'))
    liftIO $ putStr ("Cycle " ++ show i)
--    liftIO $ putStrLn (" ***** Peek at register R" ++ show regnum)
  let inps = "0 1 0 0 1 " ++ show regnum ++ " 0"
  conditional displayFullSignals $ do
    liftIO $ putStrLn ("inps = " ++ inps)
  takeInputsFromList inps
  conditional displayFullSignals $ do
    printInPorts
    printOutPorts
  s <- get
  let ps = peekList s
  let b = ps!!1  --  output b from regfile
  let bs = map current b
--  liftIO $ putStrLn ("Register value = " ++ show bs)
--  liftIO $ putStrLn ("***** Peek R" ++ show regnum ++ " = " ++ bitsHex 4 bs)
  liftIO $ putStrLn ("***** I/O fetch R" ++ show regnum ++ " = " ++ bitsHex 4 bs)
  runFormat True displayFullSignals
--  case displayFullSignals of
--    False ->  advanceFormat
--    True -> runFormat True
  advanceInPorts
  advanceOutPorts
  advancePeeks
  advanceFlagTable
  incrementCycleCount
--  s <- get
--  let i = cycleCount s
--  put (s {cycleCount = i + 1})

data ProcessorMode
  = Idle
  | Booting
  | Resetting
  | Running
  deriving (Eq, Read, Show)

startup :: StateT (SysState DriverState) IO ()
startup = do
  s <- get
  put (s {userState = Just initDriverState})
  setMode Booting
  commandLoop

commandLoop :: StateT (SysState DriverState) IO ()
commandLoop = do
  liftIO $ putStr "M1> "
  liftIO $ hFlush stdout
  xs <- liftIO getLine
  let ws = words xs
  if length ws == 0
    then m1ClockCycle
  else if ws!!0 == "help"
    then printHelp
  else if ws!!0 == "run"
    then runM1simulation
  else if ws!!0 == "cycle"
    then m1ClockCycle
  else if ws!!0 == "mem"
    then do
      let start = safeReadEltInt ws 1
      let end = safeReadEltInt ws 2
      dumpMem start end
  else if ws!!0 == "regs"
    then dumpRegFile
  else if ws!!0 == "break"
    then do
      let key = safeReadEltString ws 1
      setBreakpoint key
  else if ws!!0 == "quit"
    then do
      s <- get
      put (s {running = False})
  else liftIO $ putStrLn "Invalid command, enter help for list of commands"
  s <- get
  case running s of
    True -> commandLoop
    False -> return ()

safeReadEltInt :: [String] -> Int -> Int
safeReadEltInt ws i =
  if length ws > i && i >= 0
    then read (ws!!i)
    else 0

safeReadEltString :: [String] -> Int -> String
safeReadEltString ws i =
  if length ws > i && i >= 0
    then ws!!i
    else ""

setBreakpoint :: String -> StateT (SysState a) IO ()
setBreakpoint key = do
  s <- get
  put $ s {breakpointKey = key}

-- Dump memory from start to end address

dumpMem :: Int -> Int -> StateT (SysState DriverState) IO ()
dumpMem start end = do
  case start <= end of
    True -> do
      peekMem start
      dumpMem (start+1) end
    False -> return ()

peekMem :: Int -> StateT (SysState DriverState) IO [Bool]
peekMem addr = do
  let displayFullSignals = False
  s <- get
  let i = cycleCount s
  let inp = inPortList s
  let outp = outPortList s
  conditional displayFullSignals $ do
    liftIO $ putStrLn (take 80 (repeat '-'))
    liftIO $ putStr ("Cycle " ++ show i ++ ".  ")
--  liftIO $ putStrLn (" ***** Peek at memory address " ++ show addr)
  let inps = "0 1 0 1 0 " ++ show addr ++ " 0"
  conditional displayFullSignals $ do
    liftIO $ putStrLn ("inps = " ++ inps)
  takeInputsFromList inps
  conditional displayFullSignals $ do
    printInPorts
    printOutPorts
  s <- get
  let ps = peekList s
  let a = ps!!0  -- m_out
  let bs = map current a
  liftIO $ putStrLn ("***** I/O fetch Mem[" ++ show addr ++ "] = " ++ bitsHex 4 bs)
  runFormat True displayFullSignals
--  case displayFullSignals of
--    False -> advanceFormat
--    True -> runFormat
  advanceInPorts
  advanceOutPorts
--  advanceFormat
  advancePeeks
  advanceFlagTable
  incrementCycleCount
--  s <- get
--  let i = cycleCount s
--  put (s {cycleCount = i + 1})
  return bs

dumpRegFile :: StateT (SysState DriverState) IO ()
dumpRegFile = do
  let f i =
        case i <= 15 of
          True -> do
            peekReg i
            f (i+1)
          False -> return ()
  f 0

printHelp :: StateT (SysState a) IO ()
printHelp = do
  printLine "Commands for the M1 driver"
  printLine "  (blank)     -- perform one clock cycle"
  printLine "  cycle       -- perform one clock cycle"
  printLine "  run         -- perform clock cycles repeatedly until halt or break"
  printLine "  regs        -- display contents of the register file"
  printLine "  mem a b     -- display memory from address a to b"
  printLine "  break FLAG  -- run will stop when FLAG signal is 1 (see list below)"
  printLine "  quit        -- return to ghci or shell prompt"
  printLine "  help        -- list the commands"
  printLine "The following signals can be used as break FLAG:"
  s <- get
  printLine (concat (map ((' ':) . fst) (flagTable s)))
```

```
------------------------------------------------------------------------
-- New format
------------------------------------------------------------------------

runM1simulation :: StateT (SysState DriverState) IO ()
runM1simulation = do
  printLine "runM1simulation starting"
  simulationLooper
--  printLine "runM1simulation terminated"

simulationLooper :: StateT (SysState DriverState) IO ()
simulationLooper = do
  s <- get
  if checkFlag (flagTable s) (breakpointKey s)
      || cycleCountSinceClear s >= 10000
    then do
      clearCycleCount
      cycle <- getClockCycle
      m1ClockCycle -- display the cycle where the breakpoint is satisfied
      liftIO $ putStrLn (take 72 (repeat '-'))
      liftIO $ putStrLn ("*** Breakpoint " ++ breakpointKey s
                         ++ " in cycle " ++ show cycle ++ " ***")
      liftIO $ putStrLn (take 72 (repeat '-'))
      return ()
    else do m1ClockCycle
            s <- get
            case halted s of
              True -> do
                printLine "Processor has halted"
                return ()
              False -> simulationLooper

--  doStep []
--  runSimulation
--  runUntil initDriverState mytermpred input simoutput

-- An interactive command could request something unusual (such as a
-- register dump).  However, usually the user will wish to establishM1inputs to
-- the next cycle using the normal inputs.  This is performed by
-- establishM1inputs: perform the next action if the current input list is not
-- exhausted; otherwise go to the next mode

-- If an input list is being consumed, establishM1inputs continues as long as
-- there is data.  When the input list is exhaused, it goes to the
-- next normal mode.

-- DMA is a special case.  A DMA operation may take a number of clock
-- cycles, and it interrupts normal execution of the processor.  After
-- the DMA is finished, the previous mode is resumed.  When a DMA
-- begins, the processor mode is saved in stolenMode.  A DMA operation
-- will always run to completion; it cannot be interrupted by yet
-- another DMA.

cmdM1ClockCycle :: Command DriverState
cmdM1ClockCycle _ = m1ClockCycle

getProcessorMode :: StateT (SysState DriverState) IO ProcessorMode
getProcessorMode = do
  s <- get
  let mds = userState s
  case mds of
    Nothing -> do printLine "DriverState not defined"
                  return Idle
    Just ds -> return (processorMode ds)


m1ClockCycle :: Operation DriverState
m1ClockCycle = do
--  liftIO $ putStrLn ("m1ClockCycle starting")
  s <- get
  let i = cycleCount s
  let inp = inPortList s
  let outp = outPortList s
  liftIO $ putStrLn (take 80 (repeat '-'))
  liftIO $ putStr ("Cycle " ++ show i ++ ".  ")
  establishM1inputs
  pm <- getProcessorMode
  liftIO $ putStr (show pm)
  liftIO $ putStrLn (if halted s then "  Halted" else "")
--  printInPorts
--  printOutPorts
  runFormat True True
  advanceInPorts
  advanceOutPorts
  advancePeeks
  advanceFlagTable
  incrementCycleCount
--  s <- get
--  let i = cycleCount s
--  put (s {cycleCount = i + 1})
--  liftIO $ putStrLn ("m1ClockCycle finished")


getCurrentInputs :: StateT (SysState DriverState) IO (Maybe String)
getCurrentInputs = do
  return $ Just "0 0 0 0 0 0 0"

establishM1inputs :: StateT (SysState DriverState) IO ()
establishM1inputs = do
  mds <- getUserState
  case mds of
    Nothing -> do
      printError "establishM1inputs: empty driver state"
      return ()
    Just ds ->
      case processorMode ds of
        Booting  -> do
          inp <- getStoredInput
          case inp of
            Just x -> takeInputsFromList x
            Nothing -> do
              setMode Resetting
              establishM1inputs
        Resetting -> do
          takeInputsFromList resettingInputs
          setMode Running
        Running -> takeInputsFromList runningInputs

resettingInputs = "1 0 0 0 0 0 0"
runningInputs   = "0 0 0 0 0 0 0"
```

```
-- Each operation that requires DMA is carried out by a function that
-- supplies the required inputs, but does not use any of the input
-- lists.

-- in Driver but should be added to export list (edit Driver)
-- Don't set running to false on end of input
getStoredInput :: StateT (SysState a) IO (Maybe String)
getStoredInput = do
  s <- get
  case storedInput s of
    [] -> do
--      put $ s {running = False}  fix this
      return Nothing
    (x:xs) -> do
      put $ s {storedInput = xs}
      liftIO $ putStrLn ("getStoredInput " ++ x)
      return (Just x)

setMode :: ProcessorMode -> StateT (SysState DriverState) IO ()
setMode m = do
--  printLine ("Setting mode to " ++ show m)
  mds <- getUserState
  case mds of
    Just ds -> do
      let ds' = ds { processorMode  = m }
      s <- get
      put (s {userState = Just ds'})
      return ()
    Nothing -> do
      printError "setMode, getUserState returned Nothing"
      return ()


--------------------------------------------------------------------------------
-- M1 clock cycle
--------------------------------------------------------------------------------

{-
Every clock cycle consists of a sequence of phases:

- *choose* -- place inputs for current cycle into (currentInputString
  field of SysState).  Decide what inputs to use during the upcoming
  cycle, before it starts, and save the inputs as a string in a
  canonical format; it is placed in the currentInputString field of
  SysState.  The choice may depend on the driver state, annd/or it may
  depend on some input provided interactively by the user.  Once the
  inputs are chosen, all the signal values are fixed for the cycle
  (although what is actually output can be chosen later)

  - establish inputs - th
  - read the signals during the cycle
  - advance

The only thing that affects what happens during the cycle is the
inputs; controlling the cycle is done by establish inputs.  The middle
section just determines what is output, but not any signal values.
The advance has no choices to make; this is purely mechanical.

-}

-- selectModeInputs :: StateT (SysState DriverState) IO ()
-- selectModeInputs = do
--   s <- get
--   let ds = userState s
--   return ()
-- Similar to Driver.clockCycle but specialized for M1 driver



cmdBoot :: Command DriverState
cmdBoot _ = doBoot

doBoot :: Operation a
doBoot = do
  printLine "Booting..."
--  selectInputList "boot"

cmdReset :: Command DriverState
cmdReset _ = doReset

doReset :: Operation a
doReset = do
  printLine "Resetting"
--  selectInputList  "reset"



--------------------------------------------------------------------------------
-- Clock cycle for M1
--------------------------------------------------------------------------------

--------------------------------------------------------------------------------
-- Input signals for key stages of execution
--------------------------------------------------------------------------------

{-
I/O control inputs, copied from System
dma           1 bit    indicates stolen clock cycle
dma_store     1 bit    mem[dma_a] := dma_d
dma_fetch     1 bit    m_out = mem[dma_a]
dma_reg       1 bit    x = reg[dma_a]  (least significant 4 bits
dma_a         16 bits  address
dma_d         16 bits  data
-}


resetData :: [String]
resetData =  ["1 0 0 0 0 0 0"]

runData :: [String]
runData =  ["0 0 0 0 0 0 0"]


data DriverState = DriverState
  {
    displacement :: (Int,[Int])      -- (cycle, displacement)
  , effAddr :: [Int]                 -- effective address
  , rfloads :: [(Int,[Int],[Int])]   -- [(cycle,reg,value)]
  , memStores :: [(Int,[Int],[Int])] -- [(cycle,addr,value)]
  , jumps :: [(Int,[Int],[Int])]       -- [(cycle,jumped,pcvalue)]
  , trap :: Bool
  , processorMode :: ProcessorMode
  }                    -- has a trap just been executed?
  deriving Show

initDriverState :: DriverState
initDriverState =
  DriverState
    {
      displacement = (0, take 16 (repeat 0))
    , effAddr = take 16 (repeat 0)
    , rfloads = []
    , memStores = []
    , jumps = []
    , trap = False
    , processorMode = Idle
    }

-- Record and display the effective address

setEffAddr :: (Signal a, Static a) =>
   DriverState -> [[a]] -> DriverState
setEffAddr s [x] =
  s {effAddr = map sigInt x}

showEffAddr :: DriverState -> String
showEffAddr s = ints16hex4 (effAddr s)

-- Record and display loads to the register file

-- setRfLoad :: (Signal a, Static a) =>
--    [[a]] -> StateT (SysState DriverState) IO ()
setRfLoad :: (Signal a, Static a) => [[a]] -> StateT (SysState DriverState) IO ()
setRfLoad [r,x] = do
  c  <- getClockCycle
  s  <- get
  case userState s of
    Nothing -> return ()
    Just ds -> do
      let xs = rfloads ds
      let ds' = ds {rfloads = (c, map sigInt r, map sigInt x) : xs}
      put $ s {userState = Just ds'}


clearRfLoads :: (Signal a, Static a) =>
  DriverState -> [[a]] -> DriverState
clearRfLoads s _ = s {rfloads = []}

showRfLoads :: DriverState -> String
showRfLoads s = concat (map f (reverse (rfloads s)))
  where f (c,r,x) =
          "R" ++ show (intsInt r) ++ " := " ++  ints16hex4 x
            ++ " was loaded in cycle " ++ show c ++ "\n"
--            ++ " was loaded\n"

-- Record and display stores to the memory

setMemStore :: (Signal a, Static a) => [[a]] -> StateT (SysState DriverState) IO ()
setMemStore [a,x] = do
  c <- getClockCycle
  s <- get
  case userState s of
    Nothing -> return ()
    Just  ds -> do
      let xs = memStores ds
      let ds' = ds {memStores = (c, map sigInt a, map sigInt x) : xs}
      put $ s {userState = Just ds'}

clearMemStores :: (Signal a, Static a) =>
  DriverState -> [[a]] -> DriverState
clearMemStores s _ = s {memStores = []}

showMemStores :: DriverState -> String
showMemStores s = concat (map f (reverse (memStores s)))
  where f (c,a,x) =
          "mem[" ++ ints16hex4 a ++ "] := " ++  ints16hex4 x
            ++ " was stored in cycle " ++ show c ++ "\n"
```

```
-- When the driver discovers that a trap has executed, it uses setTrap
-- to record this in the driver state.  The termination predicate uses
-- this value to decide when to stop the simulation.

setTrap :: DriverState -> [a] -> DriverState
setTrap s _ = s {trap = True}

setDisplacement :: (Signal a, Static a) =>
   DriverState -> [[a]] -> DriverState
setDisplacement s [w] =
  s {displacement = (0, map sigInt w)}

showDisplacement :: DriverState -> String
showDisplacement s =
  let (c,d) = displacement s
--  in "displacement " ++ ints16hex4 d
--       ++ " loaded in cycle " ++ show c ++ "\n"
  in ints16hex4 d

-- The termination predicate, termpred, stops the simulation when a
-- trap instruction is executed, or after 1000 cycles.

mytermpred :: DriverState -> Bool
mytermpred s = trap s  -- || c > 10000

----------------------------------------------------------------------
-- Decoding instructions

-- When an instruction is decoded, findMnemonic returns the assembly
-- language mnemonic for the instruction, given the opcode.  The
-- opcode consists of the op field of the instruction, as well as the
-- sb field.  For RRR instructions, the op field determines the
-- instruction.  For RX instructions, the op is 15, which indicates an
-- escape to the sb field.

findMnemonic :: [[Int]] -> String
findMnemonic [opfield, bfield] =
  let op = intsInt opfield
      b = intsInt bfield
      mnemonics_RRR =
        ["add", "sub", "mul", "div",
         "cmp", "nop", "nop", "nop",
         "nop", "nop", "nop", "trap",
         "nop", "nop", "expandExp", "expandRX"]
      mnemonics_RX =
        ["lea",    "load",   "store", "jump",
         "jumpc0", "jumpc1", "jal",   "nop",
         "nop",    "nop",    "nop",   "nop",
         "nop",    "nop",    "nop",   "nop"]
  in if op==15
       then mnemonics_RX !! b
       else mnemonics_RRR !! op

--------------------------------------------------------------------------------
-- Utilities
--------------------------------------------------------------------------------

tryRead :: String -> IO (Either IOError String)
tryRead path = do
  x <- try (readFile path)
  return x

maybeRead :: String -> IO (Maybe String)
maybeRead path = do
  r <- tryRead path
  case r of
    Left e -> return Nothing
    Right x -> return (Just x)

mkFullPath :: String -> String -> String
mkFullPath prefix path =
  prefix ++ path ++ ".obj.txt"
```