The M1 Processor Circuit for Sigma16

John T. O'Donnell

October 12, 2021

Contents

1 Introduction **2**

2 Running the system **2**

2.1 Installation 2
2.2 Quick start 2
2.3 Configuration 3
2.4 An example program 3
2.5 Assemble the program 4
2.6 Run the machine code on the M1 circuit 4
2.7 Breakpoints 5
2.8 Dumping the register file and memory 6
2.9 Commands 6

3 Datapath **7**

3.1 ALU: the Arithmetic and Logic Unit 7
3.1.1 Interface to ALU 7
3.1.2 Running test data on the ALU 10
3.1.3 The ALU circuit 10
3.2 Basic register file 14
3.3 Handling R0 and R15 15
3.4 Implementing the register file with special cases 17

4 Control **19**

4.1 Control state: basic delay elements 19
4.2 Basic delay elemeent method for control 20
4.3 Enhanced delay elements for DMA and cycle stealing 20

5 System **23**

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 Running the system

2.1 Installation

- Install ghc
- The current version of M1 contains Hydra in a subdirectory, so you don't need to install it separately.

2.2 Quick start

The following commands will start M1, boot an example program, and run it to completion.

```
$ 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
```

Now press Enter repeatedly to step through the execution, once clock cycle at a time.

2.3 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.

2.4 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; see README and 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
```

2.5 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.

2.6 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.

2.7 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
```

2.8 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.

2.9 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.

3 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.

3.1 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.

3.1.1 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

3.1.2 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
```

3.1.3 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 comparitor. 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 comparitor 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 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 sum and ccnew, the new value of the condition code.

```
alu n (alua,alub,aluc) x y cc = (sum, 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.

```
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 comparitor) and it uses preprocessing of the inputs to the adder post processing of the output from the comparitor. 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.

```
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
comparing = and3 alua (inv alub) (inv aluc)  -- doing comparison, alu abc = 100
```

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.)

```
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 sum; this will give the sign.

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

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

```
(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 comparitor. 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 $x > y$.

```
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.

```
natovfl = carry           -- natural (binary) overflow if carry = 1
intovfl = xor2 carry msb  -- integer (2's comp) overflow if carry != msb
noOvfl  = inv intovfl   -- no integer overflow, integer result is ok
```

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.

```
any1 = orw sum         -- 1 if any bit in sum 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.

```
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.

```
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.

```
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
        ]
```

3.2 Basic register file

```
regfile1
  :: CBit a   -- sequential circuit with signal type a
  => Int      -- k is number of reg address bits, with 2^k registers
  -> a        -- ld.  if ld then reg[d] := x
  -> [a]      -- d is destination address, register to load
  -> [a]      -- sa is source address a, first operand to fetch
  -> [a]      -- sb is source address b, second operand to fetch
  -> a        -- x is data to load into reg[d] if ld=1
  -> (a,a)    -- (reg[sa], reg[sb]) is the readouts of two registers

regfile1 k ld d sa sb x
  | k==0 = (r,r)
  | k>0  = (a,b)
  where
    r = reg1 ld x
    (a0,b0) = regfile1 (k-1) ld0 ds sas sbs x
    (a1,b1) = regfile1 (k-1) ld1 ds sas sbs x
    (ld0,ld1) = demux1 d1 ld
    a = mux1 sa1 a0 a1
    b = mux1 sb1 b0 b1
    (d1:ds) = d
    (sa1:sas) = sa
    (sb1:sbs) = sb
```

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

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

3.3 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 `Reg[ir_d]`) also set some flags in the condition code indicating overflow and other conditions. The conditional jump instructions automatically fetch bits from R15.

There are several ways the circuit could handle R0. M1 uses the best approach: R0 does not actully have any flip flops. A load into R0 is simply discarded, and the combinational logic for fetching a register produces the value 0 when R0 is fetched. This approach 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. There are two reasons:

- The machine may need to read out both R15 and some other register at the same time.
- The machine may need to load a new value into both R15 and some other register at the same time.

Both of these requirements cannot be satisfied with the basic register file circuit. Instead, it would be necessary to use two clock cycles.

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.)

It is critically important to be able

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

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]

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]
```

3.4 Implementing the register file with special cases

Recursion is controlled by the addresses; if their lengths vary there will be a pattern match error

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

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
```

– 1-bit Register file with special cases for lowest and highest – indices (R0, cc). Sigma16 uses 4-bit addresses for d, sa, sb, – leading to 16 registers, where the special cases are R0 (the head, – i.e. lowest address) and R15 (the tail, i.e. highest address)

```
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])
```

– Recursion is based on the address words. There will be a pattern – match error if the addresses (d, sa, sb) don't all have the same – number of bits. There will also be a pattern match error if the – base case has RFtype = RFfull, as a singleton register cannot be – both R0 and R15.

– Base cases

```
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)
```

– Recursion case

```
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
```

– 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]]
```

4 Control

4.1 Control state: basic delay elements

```
st_a = dff ...
st_b = dff st_a
st_c = dff st_b
st_d = dff st_c
...

st_dispatch = dff ...
ps = demux4 op st_a

st_a0 = dff (ps!!0)  -- op=0000 indicates operation a
st_a1 = dff st_a0
st_a2 = dff st_a1

st_b0 = dff (ps!!1)  -- op=0001 indicates operation b
```

```
st_b1 = dff st_a0
st_b2 = dff st_a1

st_c0 = dff (ps!!2)  -- op=0010 indicates operation b
st_c1 = dff st_c0
st_c2 = dff st_c1
...

st_z0 = dff (ps!!15)  -- op=1111 indicates operation z
st_z1 = dff st_z0
st_z2 = dff st_z1
...
```

4.2 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

- $=c1 = orw [dff=load0, dff_{load2}]$
- $=c2 = orw [dff_{load1}]$
- $=c3 = orw [dff_{load0}, dff_{load2}]$

4.3 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.
-

5 System